

Scott Rifenbark
 Scotty's Documentation Services, INC
[<srifenbark@gmail.com>](mailto:srifenbark@gmail.com)

Copyright © 2010-2020 Linux Foundation

Permission is granted to copy, distribute and/or modify this document under the terms of the [Creative Commons Attribution-Share Alike 2.0 UK: England & Wales](#) as published by Creative Commons.

Manual Notes

- This version of the **Yocto Project Mega-Manual** is for the 3.0.1 release of the Yocto Project. To be sure you have the latest version of the manual for this release, go to the [Yocto Project documentation page](#) and select the manual from that site. Manuals from the site are more up-to-date than manuals derived from the Yocto Project released TAR files.
- If you located this manual through a web search, the version of the manual might not be the one you want (e.g. the search might have returned a manual much older than the Yocto Project version with which you are working). You can see all Yocto Project major releases by visiting the [Releases](#) page. If you need a version of this manual for a different Yocto Project release, visit the [Yocto Project documentation page](#) and select the manual set by using the "ACTIVE RELEASES DOCUMENTATION" or "DOCUMENTS ARCHIVE" pull-down menus.
- To report any inaccuracies or problems with this manual, send an email to the Yocto Project discussion group at yocto@yoctoproject.com or log into the freenode #yocto channel.

Revision History	
Revision 1.8	April 2015
Released with the Yocto Project 1.8 Release.	
Revision 2.0	October 2015
Released with the Yocto Project 2.0 Release.	
Revision 2.1	April 2016
Released with the Yocto Project 2.1 Release.	
Revision 2.2	October 2016
Released with the Yocto Project 2.2 Release.	
Revision 2.3	May 2017
Released with the Yocto Project 2.3 Release.	
Revision 2.4	October 2017
Released with the Yocto Project 2.4 Release.	
Revision 2.5	May 2018
Released with the Yocto Project 2.5 Release.	
Revision 2.6	November 2018
Released with the Yocto Project 2.6 Release.	
Revision 2.7	May 2019
Released with the Yocto Project 2.7 Release.	
Revision 3.0	October 2019
Released with the Yocto Project 3.0 Release.	
Revision 3.0.1	December 2019
Released with the Yocto Project 3.0.1 Release.	

Abstract¶

The Yocto Project Mega-Manual is a concatenation of the published Yocto Project HTML manuals along with the corresponding BitBake User Manual for the given release. The Mega-Manual exists to help users efficiently search for strings across the entire Yocto Project documentation set inclusive of the BitBake User Manual.

Yocto Project Quick Build

Yocto Project Quick Build¶

Copyright © 2010-2020 Linux Foundation

Permission is granted to copy, distribute and/or modify this document under the terms of the [Creative Commons Attribution-Share Alike 2.0 UK: England & Wales](#) as published by Creative Commons.

Abstract¶



1. Welcome!¶

Welcome! This short document steps you through the process for a typical image build using the Yocto Project. The document also introduces how to configure a build for specific hardware. You will use Yocto Project to build a reference embedded OS called Poky.

Notes

- The examples in this paper assume you are using a native Linux system running a recent Ubuntu Linux distribution. If the machine you want to use Yocto Project on to build an image ([build host](#)) is not a native Linux system, you can still perform these steps by using CROss PlatformS (CROPS) and setting up a Poky container. See the [Setting Up to Use CROss PlatformS \(CROPS\)](#) section in the Yocto Project Development Tasks Manual for more information.
- You cannot use a build host that is using the [Windows Subsystem for Linux](#) (WSL). The Yocto Project is not compatible with WSL.

If you want more conceptual or background information on the Yocto Project, see the Yocto project Overview and Concepts Manual.

2. Compatible Linux Distribution¶

Make sure your [build host](#) meets the following requirements:

- 50 Gbytes of free disk space
- Runs a supported Linux distribution (i.e. recent releases of Fedora, openSUSE, CentOS, Debian, or Ubuntu). For a list of Linux distributions that support the Yocto Project, see the [Supported Linux Distributions](#) section in the Yocto Project Reference Manual. For detailed information on preparing your build host, see the [Preparing the Build Host](#) section in the Yocto Project Development Tasks Manual.
- - Git 1.8.3.1 or greater
 - tar 1.27 or greater
 - Python 3.4.0 or greater.

If your build host does not meet any of these three listed version requirements, you can take steps to prepare the system so that you can still use the Yocto Project. See the [Required Git, tar, and Python Versions](#) section in the Yocto Project Reference Manual for information.

3. Build Host Packages¶

You must install essential host packages on your build host. The following command installs the host packages based on an Ubuntu distribution:

Note

For host package requirements on all supported Linux distributions, see the [Required Packages for the Build Host](#) section in the Yocto Project Reference Manual.

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib \
build-essential chrpath socat cpio python python3 python3-pip python3-pexpect \
xz-utils debiannutils iputils-ping python3-git python3-jinja2 libegl1-mesa libstd1.2-dev \
```

pylint3 xterm

4. Use Git to Clone Poky

Once you complete the setup instructions for your machine, you need to get a copy of the Poky repository on your build host. Use the following commands to clone the Poky repository.

```
$ git clone git://git.yoctoproject.org/poky
Cloning into 'poky'...
remote: Counting objects: 432160, done.
remote: Compressing objects: 100% (102056/102056), done.
remote: Total 432160 (delta 323116), reused 432037 (delta 323000)
Receiving objects: 100% (432160/432160), 153.81 MiB | 8.54 MiB/s, done.
Resolving deltas: 100% (323116/323116), done.
Checking connectivity... done.
```

Move to the `poky` directory and take a look at the tags:

```
$ cd poky
$ git fetch --tags
$ git tag
1.1_M1.final
1.1_M1.rc1
1.1_M1.rc2
1.1_M2.final
1.1_M2.rc1
.
.
.
yocto-2.5
yocto-2.5.1
yocto-2.5.2
yocto-2.6
yocto-2.6.1
yocto-2.6.2
yocto-2.7
yocto_1.5_M5.rc8
```

For this example, check out the branch based on the yocto-3.0.1 release:

```
$ git checkout tags/yocto-3.0.1 -b my-yocto-3.0.1
Switched to a new branch 'my-yocto-3.0.1'
```

The previous Git checkout command creates a local branch named `my-yocto-3.0.1`. The files available to you in that branch exactly match the repository's files in the "zeus" development branch at the time of the Yocto Project yocto-3.0.1 release.

For more options and information about accessing Yocto Project related repositories, see the "[Locating Yocto Project Source Files](#)" section in the Yocto Project Development Tasks Manual.

5. Building Your Image

Use the following steps to build your image. The build process creates an entire Linux distribution, including the toolchain, from source.

Note

- If you are working behind a firewall and your build host is not set up for proxies, you could encounter problems with the build process when fetching source code (e.g. fetcher failures or Git failures).
- If you do not know your proxy settings, consult your local network infrastructure resources and get that information. A good starting point could also be to check your web browser settings. Finally, you can find more information on the "[Working Behind a Network Proxy](#)" page of the Yocto Project Wiki.

1. **Initialize the Build Environment:** From within the `poky` directory, run the `oe-init-build-env` environment setup script to define Yocto Project's build environment on your build host.

```
$ cd ~/poky
$ source oe-init-build-env
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, for
example, select a different MACHINE (target hardware). See conf/local.conf
for more information as common configuration options are commented.

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

The Yocto Project has extensive documentation about OE including a reference
manual which can be found at:
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-ide-support
```

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

Among other things, the script creates the [Build Directory](#), which is `build` in this case and is located in the [Source Directory](#). After the script runs, your current working directory is set to the Build Directory. Later, when the build completes, the Build Directory contains all the files created during the build.

2. **Examine Your Local Configuration File:** When you set up the build environment, a local configuration file named `local.conf` becomes available in a `conf` subdirectory of the Build Directory. For this example, the defaults are set to build for a `qemux86` target, which is suitable for emulation. The package manager used is set to the RPM package manager.

Tip

You can significantly speed up your build and guard against fetcher failures by using mirrors. To use mirrors, add these lines to your `local.conf` file in the Build directory:

```
SSTATE_MIRRORS = "\
file:///.* http://sstate.yoctoproject.org/dev/PATH;downloadfilename=PATH \n \
file:///.* http://sstate.yoctoproject.org/2.7.2/PATH;downloadfilename=PATH \n \
file:///.* http://sstate.yoctoproject.org/3.0.1/PATH;downloadfilename=PATH \n \
"
```

The previous examples showed how to add sstate paths for Yocto Project 2.7.2, 3.0.1, and a development area. For a complete index of sstate locations, see <http://sstate.yoctoproject.org/>.

3. **Start the Build:** Continue with the following command to build an OS image for the target, which is `core-image-sato` in this example:

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

For information on using the `bitbake` command, see the "[BitBake](#)" section in the Yocto Project Overview and Concepts Manual, or see the "[BitBake Command](#)" section in the BitBake User Manual.

4. **Simulate Your Image Using QEMU:** Once this particular image is built, you can start QEMU, which is a Quick EMULATOR that ships with the Yocto Project:

```
$ runqemu qemux86
```

If you want to learn more about running QEMU, see the "[Using the Quick EMULATOR \(QEMU\)](#)" chapter in the Yocto Project Development Tasks Manual.

5. **Exit QEMU:** Exit QEMU by either clicking on the shutdown icon or by typing `Ctrl-C` in the QEMU transcript window from which you evoked QEMU.

6. Customizing Your Build for Specific Hardware¶

So far, all you have done is quickly built an image suitable for emulation only. This section shows you how to customize your build for specific hardware by adding a hardware layer into the Yocto Project development environment.

In general, layers are repositories that contain related sets of instructions and configurations that tell the Yocto Project what to do. Isolating related metadata into functionally specific layers facilitates modular development and makes it easier to reuse the layer metadata.

Note

By convention, layer names start with the string "meta-".

Follow these steps to add a hardware layer:

1. **Find a Layer:** Lots of hardware layers exist. The Yocto Project [Source Repositories](#) has many hardware layers. This example adds the [meta-altera](#) hardware layer.
2. **Clone the Layer** Use Git to make a local copy of the layer on your machine. You can put the copy in the top level of the copy of the Poky repository created earlier:

```
$ cd ~/poky
$ git clone https://github.com/kraj/meta-altera.git
Cloning into 'meta-altera'...
remote: Counting objects: 25170, done.
remote: Compressing objects: 100% (350/350), done.
remote: Total 25170 (delta 645), reused 719 (delta 538), pack-reused 24219
Receiving objects: 100% (25170/25170), 41.02 MiB | 1.64 MiB/s, done.
Resolving deltas: 100% (13385/13385), done.
Checking connectivity... done.
```

The hardware layer now exists with other layers inside the Poky reference repository on your build host as `meta-altera` and contains all the metadata needed to support hardware from Altera, which is owned by Intel.

3. **Change the Configuration to Build for a Specific Machine:** The `MACHINE` variable in the `local.conf` file specifies the machine for the build. For this example, set the `MACHINE` variable to "cyclone5". These configurations are used: <https://github.com/kraj/meta-altera/blob/master/conf/machine/cyclone5.conf>.

Note

See the "[Examine Your Local Configuration File](#)" step earlier for more information on configuring the build.

4. **Add Your Layer to the Layer Configuration File:** Before you can use a layer during a build, you must add it to your `bblayers.conf` file, which is found in the [Build Directory's](#) `conf` directory.

Use the `bitbake-layers add-layer` command to add the layer to the configuration file:

```
$ cd ~/poky/build
$ bitbake-layers add-layer ../meta-altera
NOTE: Starting bitbake server...
Parsing recipes: 100% |#####| Time: 0:00:32
Parsing of 918 .bb files complete (0 cached, 918 parsed). 1401 targets, 123 skipped, 0 masked, 0 errors.
```

You can find more information on adding layers in the "[Adding a Layer Using the bitbake-layers Script](#)" section.

Completing these steps has added the `meta-altera` layer to your Yocto Project development environment and configured it to build for the "cyclone5" machine.

Note

The previous steps are for demonstration purposes only. If you were to attempt to build an image for the "cyclone5" build, you should read the Altera README.

7. Creating Your Own General Layer¶

Maybe you have an application or specific set of behaviors you need to isolate. You can create your own general layer using the `bitbake-layers create-layer` command. The tool automates layer creation by setting up a subdirectory with a `layer.conf` configuration file, a `recipes-example` subdirectory that contains an `example.bb` recipe, a licensing file, and a README.

The following commands run the tool to create a layer named `meta-mylayer` in the `poky` directory:

```
$ cd ~/poky
$ bitbake-layers create-layer meta-mylayer
NOTE: Starting bitbake server...
Add your new layer with 'bitbake-layers add-layer meta-mylayer'
```

For more information on layers and how to create them, see the "[Creating a General Layer Using the bitbake-layers Script](#)" section in the Yocto Project Development Tasks Manual.

8. Where To Go Next¶

Now that you have experienced using the Yocto Project, you might be asking yourself "What now?" The Yocto Project has many sources of information including the website, wiki pages, and user manuals:

- **Website:** The [Yocto Project Website](#) provides background information, the latest builds, breaking news, full development documentation, and access to a rich Yocto Project Development Community into which you can tap.
- **Developer Screencast:** The [Getting Started with the Yocto Project - New Developer Screencast Tutorial](#) provides a 30-minute video created for users unfamiliar with the Yocto Project but familiar with Linux build hosts. While this screencast is somewhat dated, the introductory and fundamental concepts are useful for the beginner.
- **Yocto Project Overview and Concepts Manual:** The Yocto project Overview and Concepts Manual is a great place to start to learn about the Yocto Project. This manual introduces you to the Yocto Project and its development environment. The manual also provides conceptual information for various aspects of the Yocto Project.
- **Yocto Project Wiki:** The [Yocto Project Wiki](#) provides additional information on where to go next when ramping up with the Yocto Project, release information, project planning, and QA information.
- **Yocto Project Mailing Lists:** Related mailing lists provide a forum for discussion, patch submission and announcements. Several mailing lists exist and are grouped according to areas of concern. See the "[Mailing lists](#)" section in the Yocto Project Reference Manual for a complete list of Yocto Project mailing lists.
- **Comprehensive List of Links and Other Documentation:** The "[Links and Related Documentation](#)" section in the Yocto Project Reference Manual provides a comprehensive list of all related links and other user documentation.

Yocto Project Overview and Concepts Manual

Chapter 1. The Yocto Project Overview and Concepts Manual¶

- [1.1. Welcome](#)
- [1.2. Other Information](#)

1.1. Welcome¶

Welcome to the Yocto Project Overview and Concepts Manual! This manual introduces the Yocto Project by providing concepts, software overviews, best-known-methods (BKMs), and any other high-level introductory information suitable for a new Yocto Project user.

The following list describes what you can get from this manual:

- ***Introducing the Yocto Project:*** This chapter provides an introduction to the Yocto Project. You will learn about features and challenges of the Yocto Project, the layer model, components and tools, development methods, the [Poky](#) reference distribution, the OpenEmbedded build system workflow, and some basic Yocto terms.
- ***The Yocto Project Development Environment:*** This chapter helps you get started understanding the Yocto Project development environment. You will learn about open source, development hosts, Yocto Project source repositories, workflows using Git and the Yocto Project, a Git primer, and information about licensing.
- ***Yocto Project Concepts:*** This chapter presents various concepts regarding the Yocto Project. You can find conceptual information about components, development, cross-toolchains, and so forth.

This manual does not give you the following:

- ***Step-by-step Instructions for Development Tasks:*** Instructional procedures reside in other manuals within the Yocto Project documentation set. For example, the Yocto Project Development Tasks Manual provides examples on how to perform various development tasks. As another example, the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual contains detailed instructions on how to install an SDK, which is used to develop applications for target hardware.
- ***Reference Material:*** This type of material resides in an appropriate reference manual. For example, system variables are documented in the Yocto Project Reference Manual. As another example, the Yocto Project Board Support Package (BSP) Developer's Guide contains reference information on BSPs.
- ***Detailed Public Information Not Specific to the Yocto Project:*** For example, exhaustive information on how to use the Source Control Manager Git is better covered with Internet searches and official Git Documentation than through the Yocto Project documentation.

1.2. Other Information¶

Because this manual presents information for many different topics, supplemental information is recommended for full comprehension. For additional introductory information on the Yocto Project, see the [Yocto Project Website](#). If you want to build an image with no knowledge of Yocto Project as a way of quickly testing it out, see the Yocto Project Quick Build document. For a comprehensive list of links and other documentation, see the "[Links and Related Documentation](#)" section in the Yocto Project Reference Manual.

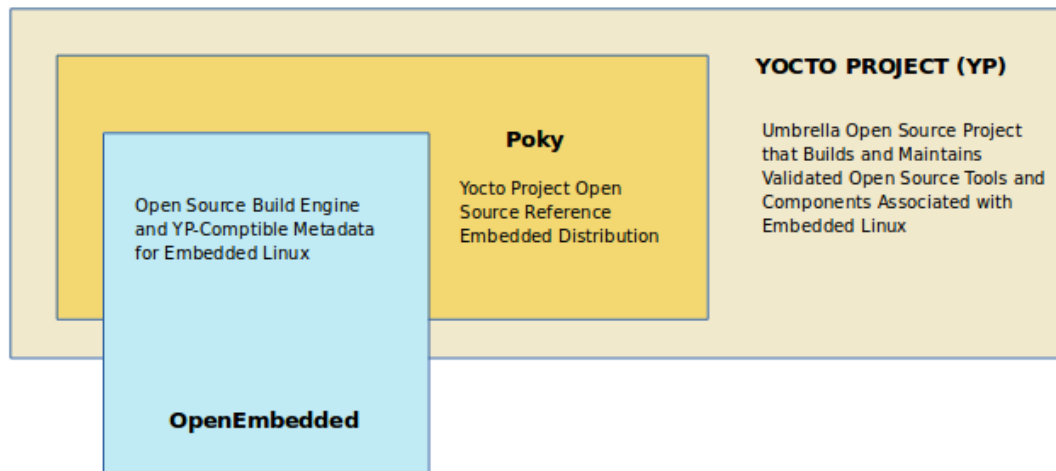
Chapter 2. Introducing the Yocto Project¶

- [2.1. What is the Yocto Project?](#)
 - [2.1.1. Features](#)
 - [2.1.2. Challenges](#)
- [2.2. The Yocto Project Layer Model](#)
- [2.3. Components and Tools](#)
 - [2.3.1. Development Tools](#)
 - [2.3.2. Production Tools](#)
 - [2.3.3. Open-Embedded Build System Components](#)
 - [2.3.4. Reference Distribution \(Poky\)](#)
 - [2.3.5. Packages for Finished Targets](#)
 - [2.3.6. Archived Components](#)
- [2.4. Development Methods](#)
- [2.5. Reference Embedded Distribution \(Poky\)](#)
- [2.6. The OpenEmbedded Build System Workflow](#)
- [2.7. Some Basic Terms](#)

2.1. What is the Yocto Project?¶

The Yocto Project is an open source collaboration project that helps developers create custom Linux-based systems that are designed for embedded products regardless of the product's hardware architecture. Yocto Project provides a flexible toolset and a development environment that allows embedded device developers across the world to collaborate through shared technologies, software stacks, configurations, and best practices used to create these tailored Linux images.

Thousands of developers worldwide have discovered that Yocto Project provides advantages in both systems and applications development, archival and management benefits, and customizations used for speed, footprint, and memory utilization. The project is a standard when it comes to delivering embedded software stacks. The project allows software customizations and build interchange for multiple hardware platforms as well as software stacks that can be maintained and scaled.



For further introductory information on the Yocto Project, you might be interested in this [article](#) by Drew Moseley and in this short introductory [video](#).

The remainder of this section overviews advantages and challenges tied to the Yocto Project.

2.1.1. Features¶

The following list describes features and advantages of the Yocto Project:

- **Widely Adopted Across the Industry:** Semiconductor, operating system, software, and service vendors exist whose products and services adopt and support the Yocto Project. For a look at the Yocto Project community and the companies involved with the Yocto Project, see the "COMMUNITY" and "ECOSYSTEM" tabs on the [Yocto Project](#) home page.
 - **Architecture Agnostic:** Yocto Project supports Intel, ARM, MIPS, AMD, PPC and other architectures. Most ODMs, OSVs, and chip vendors create and supply BSPs that support their hardware. If you have custom silicon, you can create a BSP that supports that architecture.
- Aside from lots of architecture support, the Yocto Project fully supports a wide range of device emulation through the Quick EMulator (QEMU).
- **Images and Code Transfer Easily:** Yocto Project output can easily move between architectures without moving to new development environments. Additionally, if you have used the Yocto Project to create an image or application and you find yourself not able to support it, commercial Linux vendors such as Wind River, Mentor Graphics, Timesys, and ENEA could take it and provide ongoing support. These vendors have offerings that are built using the Yocto Project.
 - **Flexibility:** Corporations use the Yocto Project many different ways. One example is to create an internal Linux distribution as a code base the corporation can use across multiple product groups. Through customization and layering, a project group can leverage the base Linux distribution to create a distribution that works for their product needs.
 - **Ideal for Constrained Embedded and IoT devices:** Unlike a full Linux distribution, you can use the Yocto Project to create exactly what you need for embedded devices. You only add the feature support or packages that you absolutely need for the device. For devices that have display hardware, you can use available system components such as X11, GTK+, Qt, Clutter, and SDL (among others) to create a rich user experience. For devices that do not have a display or where you want to use alternative UI frameworks, you can choose to not install these components.
 - **Comprehensive Toolchain Capabilities:** Toolchains for supported architectures satisfy most use cases. However, if your hardware supports features that are not part of a standard toolchain, you can easily customize that toolchain through specification of platform-specific tuning parameters. And, should you need to use a third-party toolchain, mechanisms built into the Yocto Project allow for that.
 - **Mechanism Rules Over Policy:** Focusing on mechanism rather than policy ensures that you are free to set policies based on the needs of your design instead of adopting decisions enforced by some system software provider.
 - **Uses a Layer Model:** The Yocto Project [layer infrastructure](#) groups related functionality into separate bundles. You can incrementally add these grouped functionalities to your project as needed. Using layers to isolate and group functionality reduces project complexity and redundancy, allows you to easily extend the system, make customizations, and keep functionality organized.
 - **Supports Partial Builds:** You can build and rebuild individual packages as needed. Yocto Project accomplishes this through its [shared-state cache](#) (sstate) scheme. Being able to build and debug components individually eases project development.
 - **Releases According to a Strict Schedule:** Major releases occur on a [six-month cycle](#) predictably in October and April. The most recent two releases support point releases to address common vulnerabilities and exposures. This predictability is crucial for projects based on the Yocto Project and allows development teams to plan activities.
 - **Rich Ecosystem of Individuals and Organizations:** For open source projects, the value of community is very important. Support forums, expertise, and active developers who continue to push the Yocto Project forward are readily available.

- **Binary Reproducibility:** The Yocto Project allows you to be very specific about dependencies and achieves very high percentages of binary reproducibility (e.g. 99.8% for `core-image-minimal`). When distributions are not specific about which packages are pulled in and in what order to support dependencies, other build systems can arbitrarily include packages.
- **License Manifest:** The Yocto Project provides a [license manifest](#) for review by people who need to track the use of open source licenses (e.g. legal teams).

2.1.2. Challenges¶

The following list presents challenges you might encounter when developing using the Yocto Project:

- **Steep Learning Curve:** The Yocto Project has a steep learning curve and has many different ways to accomplish similar tasks. It can be difficult to choose how to proceed when varying methods exist by which to accomplish a given task.
- **Understanding What Changes You Need to Make For Your Design Requires Some Research:** Beyond the simple tutorial stage, understanding what changes need to be made for your particular design can require a significant amount of research and investigation. For information that helps you transition from trying out the Yocto Project to using it for your project, see the "[What I wish I'd Known](#)" and "[Transitioning to a Custom Environment for Systems Development](#)" documents on the Yocto Project website.
- **Project Workflow Could Be Confusing:** The [Yocto Project workflow](#) could be confusing if you are used to traditional desktop and server software development. In a desktop development environment, mechanisms exist to easily pull and install new packages, which are typically pre-compiled binaries from servers accessible over the Internet. Using the Yocto Project, you must modify your configuration and rebuild to add additional packages.
- **Working in a Cross-Build Environment Can Feel Unfamiliar:** When developing code to run on a target, compilation, execution, and testing done on the actual target can be faster than running a BitBake build on a development host and then deploying binaries to the target for test. While the Yocto Project does support development tools on the target, the additional step of integrating your changes back into the Yocto Project build environment would be required. Yocto Project supports an intermediate approach that involves making changes on the development system within the BitBake environment and then deploying only the updated packages to the target.

The Yocto Project [OpenEmbedded build system](#) produces packages in standard formats (i.e. RPM, DEB, IPK, and TAR). You can deploy these packages into the running system on the target by using utilities on the target such as `rpm` or `ipk`.

- **Initial Build Times Can be Significant:** Long initial build times are unfortunately unavoidable due to the large number of packages initially built from scratch for a fully functioning Linux system. Once that initial build is completed, however, the shared-state (sstate) cache mechanism Yocto Project uses keeps the system from rebuilding packages that have not been "touched" since the last build. The sstate mechanism significantly reduces times for successive builds.

2.2. The Yocto Project Layer Model¶

The Yocto Project's "Layer Model" is a development model for embedded and IoT Linux creation that distinguishes the Yocto Project from other simple build systems. The Layer Model simultaneously supports collaboration and customization. Layers are repositories that contain related sets of instructions that tell the [OpenEmbedded build system](#) what to do. You can collaborate, share, and reuse layers.

Layers can contain changes to previous instructions or settings at any time. This powerful override capability is what allows you to customize previously supplied collaborative or community layers to suit your product requirements.

You use different layers to logically separate information in your build. As an example, you could have BSP, GUI, distro configuration, middleware, or application layers. Putting your entire build into one layer limits and complicates future customization and reuse. Isolating information into layers, on the other hand, helps simplify future customizations and reuse. You might find it tempting to keep everything in one layer when working on a single project. However, the more modular your Metadata, the easier it is to cope with future changes.

Notes

- Use Board Support Package (BSP) layers from silicon vendors when possible.
- Familiarize yourself with the [Yocto Project curated layer index](#) or the [OpenEmbedded layer index](#). The latter contains more layers but they are less universally validated.
- Layers support the inclusion of technologies, hardware components, and software components. The [Yocto Project Compatible](#) designation provides a minimum level of standardization that contributes to a strong ecosystem. "YP Compatible" is applied to appropriate products and software components such as BSPs, other OE-compatible layers, and related open-source projects, allowing the producer to use Yocto Project badges and branding assets.

To illustrate how layers are used to keep things modular, consider machine customizations. These types of customizations typically reside in a special layer, rather than a general layer, called a BSP Layer. Furthermore, the machine customizations should be isolated from recipes and Metadata that support a new GUI environment, for example. This situation gives you a couple of layers: one for the machine configurations, and one for the GUI environment. It is important to understand, however, that the BSP layer can still make machine-specific additions to recipes within the GUI environment layer without polluting the GUI layer itself with those machine-specific changes. You can accomplish this through a recipe that is a BitBake append (`.bbappend`) file, which is described later in this section.

Note

For general information on BSP layer structure, see the [Yocto Project Board Support Packages \(BSP\) Developer's Guide](#).

The [Source Directory](#) contains both general layers and BSP layers right out of the box. You can easily identify layers that ship with a Yocto Project release in the Source Directory by their names. Layers typically have names that begin with the string `meta-`.

Note

It is not a requirement that a layer name begin with the prefix `meta-`, but it is a commonly accepted standard in the Yocto Project community.

For example, if you were to examine the [tree view](#) of the poky repository, you will see several layers: `meta`, `meta-skeleton`, `meta-selftest`, `meta-poky`, and `meta-yocto-bsp`. Each of these repositories represents a distinct layer.

For procedures on how to create layers, see the "[Understanding and Creating Layers](#)" section in the Yocto Project Development Tasks Manual.

2.3. Components and Tools¶

The Yocto Project employs a collection of components and tools used by the project itself, by project developers, and by those using the Yocto Project. These components and tools are open source projects and metadata that are separate from the reference distribution (Poky) and the [OpenEmbedded build system](#). Most of the components and tools are downloaded separately.

This section provides brief overviews of the components and tools associated with the Yocto Project.

2.3.1. Development Tools¶

The following list consists of tools that help you develop images and applications using the Yocto Project:

- **CROPS:** CROPS is an open source, cross-platform development framework that leverages [Docker Containers](#). CROPS provides an easily managed, extensible environment that allows you to build binaries for a variety of architectures on Windows, Linux and Mac OS X hosts.
- **devtool:** This command-line tool is available as part of the extensible SDK (eSDK) and is its cornerstone. You can use `devtool` to help build, test, and package software within the eSDK. You can use the tool to optionally integrate what you build into an image built by the OpenEmbedded build system.

The `devtool` command employs a number of sub-commands that allow you to add, modify, and upgrade recipes. As with the OpenEmbedded build system, "recipes" represent software packages within `devtool`. When you use `devtool add`, a recipe is automatically created. When you use `devtool modify`, the specified existing recipe is used in order to determine where to get the source code and how to patch it. In both cases, an environment is set up so that when you build the recipe a source tree that is under your control is used in order to allow you to make changes to the source as desired. By default, both new recipes and the source go into a "workspace" directory under the eSDK. The `devtool upgrade` command updates an existing recipe so that you can build it for an updated set of source files.

You can read about the `devtool` workflow in the Yocto Project Application Development and Extensible Software Development Kit (eSDK) Manual in the "[Using devtool in Your SDK Workflow](#)" section.

- **Extensible Software Development Kit (eSDK):** The eSDK provides a cross-development toolchain and libraries tailored to the contents of a specific image. The eSDK makes it easy to add new applications and libraries to an image, modify the source for an existing component, test changes on the target hardware, and integrate into the rest of the OpenEmbedded build system. The eSDK gives you a toolchain experience supplemented with the powerful set of `devtool` commands tailored for the Yocto Project environment.

For information on the eSDK, see the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) Manual.

- **Toaster:** Toaster is a web interface to the Yocto Project OpenEmbedded build system. Toaster allows you to configure, run, and view information about builds. For information on Toaster, see the Toaster User Manual.

2.3.2. Production Tools¶

The following list consists of tools that help production related activities using the Yocto Project:

- **Auto Upgrade Helper:** This utility when used in conjunction with the [OpenEmbedded build system](#) (BitBake and OE-Core) automatically generates upgrades for recipes that are based on new versions of the recipes published upstream.
- **Recipe Reporting System:** The Recipe Reporting System tracks recipe versions available for Yocto Project. The main purpose of the system is to help you manage the recipes you maintain and to offer a dynamic overview of the project. The Recipe Reporting System is built on top of the [OpenEmbedded Layer Index](#), which is a website that indexes OpenEmbedded-Core layers.
- **Patchwork:** [Patchwork](#) is a fork of a project originally started by [Oz Labs](#). The project is a web-based tracking system designed to streamline the process of bringing contributions into a project. The Yocto Project uses Patchwork as an organizational tool to handle patches, which number in the thousands for every release.
- **AutoBuilder:** AutoBuilder is a project that automates build tests and quality assurance (QA). By using the public AutoBuilder, anyone can determine the status of the current "master" branch of Poky.

Note

AutoBuilder is based on [buildbot](#).

A goal of the Yocto Project is to lead the open source industry with a project that automates testing and QA procedures. In doing so, the project encourages a development community that publishes QA and test plans, publicly demonstrates QA and test plans, and encourages development of tools that automate and test and QA procedures for the benefit of the development community.

You can learn more about the AutoBuilder used by the Yocto Project [here](#).

- **Cross-Prelink:** Prelinking is the process of pre-computing the load addresses and link tables generated by the dynamic linker as compared to doing this at runtime. Doing this ahead of time results in performance improvements when the application is launched and reduced memory usage for libraries shared by many applications.

Historically, cross-prelink is a variant of prelink, which was conceived by [Jakub Jelinek](#) a number of years ago. Both prelink and cross-prelink are maintained in the same repository albeit on separate branches. By providing an emulated runtime dynamic linker (i.e. `glibc`-derived `ld.so` emulation), the cross-prelink project extends the prelink software's ability to prelink a sysroot environment. Additionally, the cross-prelink software enables the ability to work in sysroot style environments.

The dynamic linker determines standard load address calculations based on a variety of factors such as mapping addresses, library usage, and library function conflicts. The prelink tool uses this information, from the dynamic linker, to determine unique load addresses for executable and linkable format (ELF) binaries that are shared libraries and dynamically linked. The prelink tool modifies these ELF binaries with the pre-computed information. The result is faster loading and often lower memory consumption because more of the library code can be re-used from shared Copy-On-Write (COW) pages.

The original upstream prelink project only supports running prelink on the end target device due to the reliance on the target device's dynamic linker. This restriction causes issues when developing a cross-compiled system. The cross-prelink adds a synthesized dynamic loader that runs on the host, thus permitting cross-prelinking without ever having to run on a read-write target filesystem.

- **Pseudo:** Pseudo is the Yocto Project implementation of [fakeroot](#), which is used to run commands in an environment that seemingly has root privileges.

During a build, it can be necessary to perform operations that require system administrator privileges. For example, file ownership or permissions might need definition. Pseudo is a tool that you can either use directly or through the environment variable `LD_PRELOAD`. Either method allows these operations to succeed as if system administrator privileges exist even when they do not.

You can read more about Pseudo in the "[Fakeroot and Pseudo](#)" section.

2.3.3. Open-Embedded Build System Components¶

The following list consists of components associated with the [OpenEmbedded build system](#):

- **BitBake:** BitBake is a core component of the Yocto Project and is used by the OpenEmbedded build system to build images. While BitBake is key to the build system, BitBake is maintained separately from the Yocto Project.

BitBake is a generic task execution engine that allows shell and Python tasks to be run efficiently and in parallel while working within complex inter-task dependency constraints. In short, BitBake is a build engine that works through recipes written in a specific format in order to perform sets of tasks.

You can learn more about BitBake in the BitBake User Manual.

- **OpenEmbedded-Core:** OpenEmbedded-Core (OE-Core) is a common layer of metadata (i.e. recipes, classes, and associated files) used by OpenEmbedded-derived systems, which includes the Yocto Project. The Yocto Project and the OpenEmbedded Project both maintain the OpenEmbedded-Core. You can find the OE-Core metadata in the Yocto Project [Source Repositories](#).

Historically, the Yocto Project integrated the OE-Core metadata throughout the Yocto Project source repository reference system (Poky). After Yocto Project Version 1.0, the Yocto Project and OpenEmbedded agreed to work together and share a common core set of metadata (OE-Core), which contained much of the functionality previously found in Poky. This collaboration achieved a long-standing OpenEmbedded objective for having a more tightly controlled and quality-assured core. The results also fit well with the Yocto Project objective of achieving a smaller number of fully featured tools as compared to many different ones.

Sharing a core set of metadata results in Poky as an integration layer on top of OE-Core. You can see that in this [figure](#). The Yocto Project combines various components such as BitBake, OE-Core, script "glue", and documentation for its build system.

2.3.4. Reference Distribution (Poky)¶

Poky is the Yocto Project reference distribution. It contains the [Open-Embedded build system](#) (BitBake and OE-Core) as well as a set of metadata to get you started building your own distribution. See the [figure](#) in "What is the Yocto Project?" section for an illustration that shows Poky and its relationship with other parts of the Yocto Project.

To use the Yocto Project tools and components, you can download (`clone`) Poky and use it to bootstrap your own distribution.

Note

Poky does not contain binary files. It is a working example of how to build your own custom Linux distribution from source.

You can read more about Poky in the "[Reference Embedded Distribution \(Poky\)](#)" section.

2.3.5. Packages for Finished Targets¶

The following lists components associated with packages for finished targets:

- **Matchbox:** Matchbox is an Open Source, base environment for the X Window System running on non-desktop, embedded platforms such as handhelds, set-top boxes, kiosks, and anything else for which screen space, input mechanisms, or system resources are limited.

Matchbox consists of a number of interchangeable and optional applications that you can tailor to a specific, non-desktop platform to enhance usability in constrained environments.

You can find the Matchbox source in the Yocto Project [Source Repositories](#).

- **Opkg** Open PacKaGe management (opkg) is a lightweight package management system based on the itsy package (ipkg) management system. Opkg is written in C and resembles Advanced Package Tool (APT) and Debian Package (dpkg) in operation.

Opkg is intended for use on embedded Linux devices and is used in this capacity in the [OpenEmbedded](#) and [OpenWrt](#) projects, as well as the Yocto Project.

Note

As best it can, opkg maintains backwards compatibility with ipkg and conforms to a subset of Debian's policy manual regarding control files.

2.3.6. Archived Components¶

The Build Appliance is a virtual machine image that enables you to build and boot a custom embedded Linux image with the Yocto Project using a non-Linux development system.

Historically, the Build Appliance was the second of three methods by which you could use the Yocto Project on a system that was not native to Linux.

1. **Hob:** Hob, which is now deprecated and is no longer available since the 2.1 release of the Yocto Project provided a rudimentary, GUI-based interface to the Yocto Project. Toaster has fully replaced Hob.
2. **Build Appliance:** Post Hob, the Build Appliance became available. It was never recommended that you use the Build Appliance as a day-to-day production development environment with the Yocto Project. Build Appliance was useful as a way to try out development in the Yocto Project environment.
3. **CROPS:** The final and best solution available now for developing using the Yocto Project on a system not native to Linux is with [CROPS](#).

2.4. Development Methods¶

The Yocto Project development environment usually involves a [Build Host](#) and target hardware. You use the Build Host to build images and develop applications, while you use the target hardware to test deployed software.

This section provides an introduction to the choices or development methods you have when setting up your Build Host. Depending on the your particular workflow preference and the type of operating system your Build Host runs, several choices exist that allow you to use the Yocto Project.

Note

For additional detail about the Yocto Project development environment, see the "[The Yocto Project Development Environment](#)" chapter.

- **Native Linux Host:** By far the best option for a Build Host. A system running Linux as its native operating system allows you to develop software by directly using the [BitBake](#) tool. You can accomplish all aspects of development from a familiar shell of a supported Linux distribution.

For information on how to set up a Build Host on a system running Linux as its native operating system, see the "[Setting Up a Native Linux Host](#)" section in the Yocto Project Development Tasks Manual.

- **CROss PlatformS (CROPS):** Typically, you use [CROPS](#), which leverages [Docker Containers](#), to set up a Build Host that is not running Linux (e.g. Microsoft® Windows™ or macOS®).

Note

You can, however, use CROPS on a Linux-based system.

CROPS is an open source, cross-platform development framework that provides an easily managed, extensible environment for building binaries targeted for a variety of architectures on Windows, macOS, or Linux hosts. Once the Build Host is set up using CROPS, you can prepare a shell environment to mimic that of a shell being used on a system natively running Linux.

For information on how to set up a Build Host with CROPS, see the "[Setting Up to Use CROss PlatformS \(CROPS\)](#)" section in the Yocto Project Development Tasks Manual.

- **Toaster:** Regardless of what your Build Host is running, you can use Toaster to develop software using the Yocto Project. Toaster is a web interface to the Yocto Project's [Open-Embedded build system](#). The interface enables you to configure and run your builds. Information about builds is collected and stored in a database. You can use Toaster to configure and start builds on multiple remote build servers.

For information about and how to use Toaster, see the Toaster User Manual.

2.5. Reference Embedded Distribution (Poky)¶

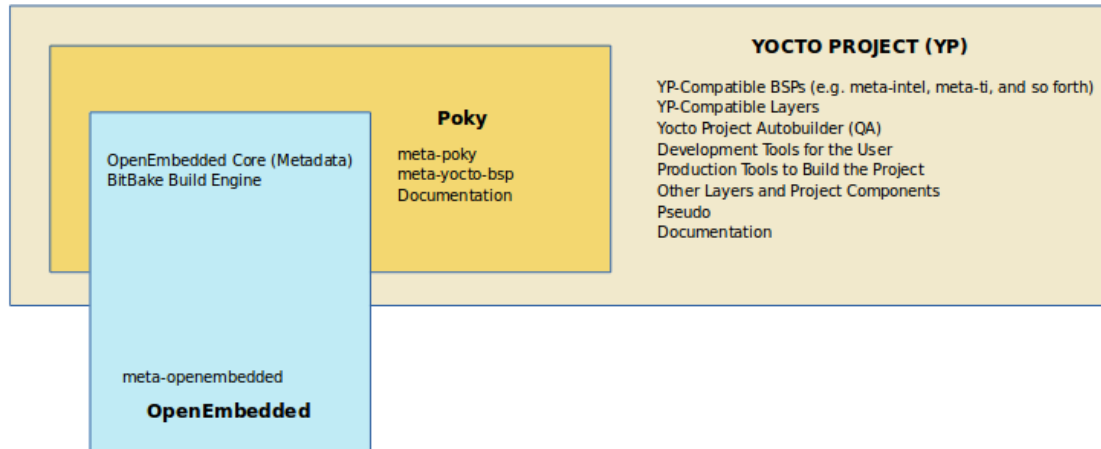
"Poky", which is pronounced **Pock**-ee, is the name of the Yocto Project's reference distribution or Reference OS Kit. Poky contains the [OpenEmbedded Build System](#) ([BitBake](#) and [OpenEmbedded-Core](#)) as well as a set of [metadata](#) to get you started building your own distro. In other words, Poky is a base specification of the functionality needed for a typical embedded system as well as the components from the Yocto Project that allow you to build a distribution into a usable binary image.

Poky is a combined repository of BitBake, OpenEmbedded-Core (which is found in `meta`), `meta-poky`, `meta-yocto-bsp`, and documentation provided all together and known to work well together. You can view these items that make up the Poky repository in the [Source Repositories](#).

Note

If you are interested in all the contents of the poky Git repository, see the "[Top-Level Core Components](#)" section in the Yocto Project Reference Manual.

The following figure illustrates what generally comprises Poky:



- BitBake is a task executor and scheduler that is the heart of the OpenEmbedded build system.
- `meta-poky`, which is Poky-specific metadata.
- `meta-yocto-bsp`, which are Yocto Project-specific Board Support Packages (BSPs).
- OpenEmbedded-Core (OE-Core) metadata, which includes shared configurations, global variable definitions, shared classes, packaging, and recipes. Classes define the encapsulation and inheritance of build logic. Recipes are the logical units of software and images to be built.
- Documentation, which contains the Yocto Project source files used to make the set of user manuals.

Note

While Poky is a "complete" distribution specification and is tested and put through QA, you cannot use it as a product "out of the box" in its current form.

To use the Yocto Project tools, you can use Git to clone (download) the Poky repository then use your local copy of the reference distribution to bootstrap your own distribution.

Note

Poky does not contain binary files. It is a working example of how to build your own custom Linux distribution from source.

Poky has a regular, well established, six-month release cycle under its own version. Major releases occur at the same time major releases (point releases) occur for the Yocto Project, which are typically in the Spring and Fall. For more information on the Yocto Project release schedule and cadence, see the "[Yocto Project Releases and the Stable Release Process](#)" chapter in the Yocto Project Reference Manual.

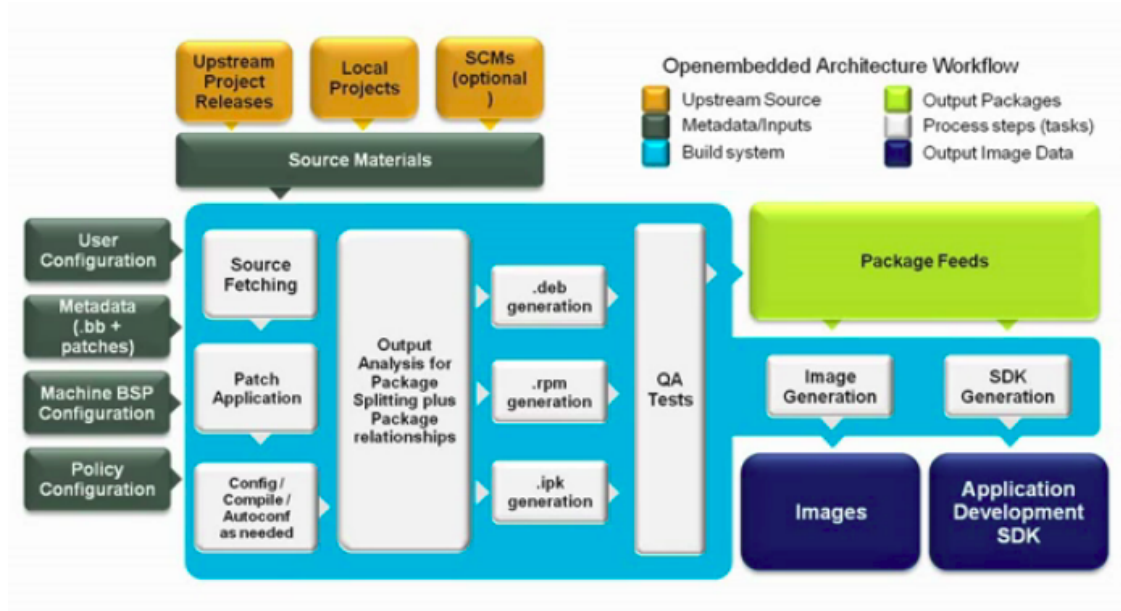
Much has been said about Poky being a "default configuration." A default configuration provides a starting image footprint. You can use Poky out of the box to create an image ranging from a shell-accessible minimal image all the way up to a Linux Standard Base-compliant image that uses a GNOME Mobile and Embedded (GMAE) based reference user interface called Sato.

One of the most powerful properties of Poky is that every aspect of a build is controlled by the metadata. You can use metadata to augment these base image types by adding metadata [layers](#) that extend functionality. These layers can provide, for example, an additional software stack for an image type, add a board support package (BSP) for additional hardware, or even create a new image type.

Metadata is loosely grouped into configuration files or package recipes. A recipe is a collection of non-executable metadata used by BitBake to set variables or define additional build-time tasks. A recipe contains fields such as the recipe description, the recipe version, the license of the package and the upstream source repository. A recipe might also indicate that the build process uses autotools, make, distutils or any other build process, in which case the basic functionality can be defined by the classes it inherits from the OE-Core layer's class definitions in `./meta/classes`. Within a recipe you can also define additional tasks as well as task prerequisites. Recipe syntax through BitBake also supports both `_prepend` and `_append` operators as a method of extending task functionality. These operators inject code into the beginning or end of a task. For information on these BitBake operators, see the "[Appending and Prepending \(Override Style Syntax\)](#)" section in the BitBake User's Manual.

2.6. The OpenEmbedded Build System Workflow¶

The [OpenEmbedded build system](#) uses a "workflow" to accomplish image and SDK generation. The following figure overviews that workflow:



Following is a brief summary of the "workflow":

1. Developers specify architecture, policies, patches and configuration details.
2. The build system fetches and downloads the source code from the specified location. The build system supports standard methods such as tarballs or source code repositories systems such as Git.
3. Once source code is downloaded, the build system extracts the sources into a local work area where patches are applied and common steps for configuring and compiling the software are run.
4. The build system then installs the software into a temporary staging area where the binary package format you select (DEB, RPM, or IPK) is used to roll up the software.
5. Different QA and sanity checks run throughout entire build process.
6. After the binaries are created, the build system generates a binary package feed that is used to create the final root file image.
7. The build system generates the file system image and a customized Extensible SDK (eSDK) for application development in parallel.

For a very detailed look at this workflow, see the "[OpenEmbedded Build System Concepts](#)" section.

2.7. Some Basic Terms¶

It helps to understand some basic fundamental terms when learning the Yocto Project. Although a list of terms exists in the "[Yocto Project Terms](#)" section of the Yocto Project Reference Manual, this section provides the definitions of some terms helpful for getting started:

- **Configuration Files:** Files that hold global definitions of variables, user-defined variables, and hardware configuration information. These files tell the [Open-Embedded build system](#) what to build and what to put into the image to support a particular platform.
- **Extensible Software Development Kit (eSDK):** A custom SDK for application developers. This eSDK allows developers to incorporate their library and programming changes back into the image to make their code available to other application developers. For information on the eSDK, see the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.
- **Layer:** A collection of related recipes. Layers allow you to consolidate related metadata to customize your build. Layers also isolate information used when building for multiple architectures. Layers are hierarchical in their ability to override previous specifications. You can include any number of available layers from the Yocto Project and customize the build by adding your layers after them. You can search the Layer Index for layers used within Yocto Project.

For more detailed information on layers, see the "[Understanding and Creating Layers](#)" section in the Yocto Project Development Tasks Manual. For a discussion specifically on BSP Layers, see the "[BSP Layers](#)" section in the Yocto Project Board Support Packages (BSP) Developer's Guide.

- **Metadata:** A key element of the Yocto Project is the Metadata that is used to construct a Linux distribution and is contained in the files that the OpenEmbedded build system parses when building an image. In general, Metadata includes recipes, configuration files, and other information that refers to the build instructions themselves, as well as the data used to control what things get built and the effects of the build. Metadata also includes commands and data used to indicate what versions of software are used, from where they are obtained, and changes or additions to the software itself (patches or auxiliary files) that are used to fix bugs or customize the software for use in a particular situation. OpenEmbedded-Core is an important set of validated metadata.

- **OpenEmbedded Build System:** The terms "BitBake" and "build system" are sometimes used for the OpenEmbedded Build System.

BitBake is a task scheduler and execution engine that parses instructions (i.e. recipes) and configuration data. After a parsing phase, BitBake creates a dependency tree to order the compilation, schedules the compilation of the included code, and finally executes the building of the specified custom Linux image (distribution). BitBake is similar to the `make` tool.

During a build process, the build system tracks dependencies and performs a native or cross-compilation of the package. As a first step in a cross-build setup, the framework attempts to create a cross-compiler toolchain (i.e. Extensible SDK) suited for the target platform.

- **OpenEmbedded-Core (OE-Core):** OE-Core is metadata comprised of foundation recipes, classes, and associated files that are meant to be common among many different OpenEmbedded-derived systems, including the Yocto Project. OE-Core is a curated subset of an original repository developed by the OpenEmbedded community that has been pared down into a smaller, core set of continuously validated recipes. The result is a tightly controlled and quality-assured core set of recipes.

You can see the Metadata in the `meta` directory of the Yocto Project [Source Repositories](#).

- **Packages:** In the context of the Yocto Project, this term refers to a recipe's packaged output produced by BitBake (i.e. a "baked recipe"). A package is generally the compiled binaries produced from the recipe's sources. You "bake" something by running it through BitBake.

It is worth noting that the term "package" can, in general, have subtle meanings. For example, the packages referred to in the ["Required Packages for the Build Host"](#) section in the Yocto Project Reference Manual are compiled binaries that, when installed, add functionality to your Linux distribution.

Another point worth noting is that historically within the Yocto Project, recipes were referred to as packages - thus, the existence of several BitBake variables that are seemingly mis-named, (e.g. `PR`, `PV`, and `PE`).

- **Poky:** Poky is a reference embedded distribution and a reference test configuration. Poky provides the following:
 - A base-level functional distro used to illustrate how to customize a distribution.
 - A means by which to test the Yocto Project components (i.e. Poky is used to validate the Yocto Project).
 - A vehicle through which you can download the Yocto Project.

Poky is not a product level distro. Rather, it is a good starting point for customization.

Note

Poky is an integration layer on top of OE-Core.

- **Recipe:** The most common form of metadata. A recipe contains a list of settings and tasks (i.e. instructions) for building packages that are then used to build the binary image. A recipe describes where you get source code and which patches to apply. Recipes describe dependencies for libraries or for other recipes as well as configuration and compilation options. Related recipes are consolidated into a layer.

Chapter 3. The Yocto Project Development Environment¶

- [3.1. Open Source Philosophy](#)
- [3.2. The Development Host](#)
- [3.3. Yocto Project Source Repositories](#)
- [3.4. Git Workflows and the Yocto Project](#)
- [3.5. Git](#)
 - [3.5.1. Repositories, Tags, and Branches](#)
 - [3.5.2. Basic Commands](#)
- [3.6. Licensing](#)

This chapter takes a look at the Yocto Project development environment. The chapter provides Yocto Project Development environment concepts that help you understand how work is accomplished in an open source environment, which is very different as compared to work accomplished in a closed, proprietary environment.

Specifically, this chapter addresses open source philosophy, source repositories, workflows, Git, and licensing.

3.1. Open Source Philosophy¶

Open source philosophy is characterized by software development directed by peer production and collaboration through an active community of developers. Contrast this to the more standard centralized development models used by commercial software companies where a finite set of developers produces a product for sale using a defined set of procedures that ultimately result in an end product whose architecture and source material are closed to the public.

Open source projects conceptually have differing concurrent agendas, approaches, and production. These facets of the development process can come from anyone in the public (community) who has a stake in the software project. The open source environment contains new copyright, licensing, domain, and consumer issues that differ from the more traditional development environment. In an open source environment, the end product, source material, and documentation are all available to the public at no cost.

A benchmark example of an open source project is the Linux kernel, which was initially conceived and created by Finnish computer science student Linus Torvalds in 1991. Conversely, a good example of a non-open source project is the Windows® family of operating systems developed by Microsoft® Corporation.

Wikipedia has a good historical description of the Open Source Philosophy [here](#). You can also find helpful information on how to participate in the Linux Community [here](#).

3.2. The Development Host¶

A development host or [build host](#) is key to using the Yocto Project. Because the goal of the Yocto Project is to develop images or applications that run on embedded hardware, development of those images and applications generally takes place on a system not intended to run the software - the development host.

You need to set up a development host in order to use it with the Yocto Project. Most find that it is best to have a native Linux machine function as the development host. However, it is possible to use a system that does not run Linux as its operating system as your development host. When you have a Mac or Windows-based system, you can set it up as the development host by using [CROPS](#), which

leverages [Docker Containers](#). Once you take the steps to set up a CROPS machine, you effectively have access to a shell environment that is similar to what you see when using a Linux-based development host. For the steps needed to set up a system using CROPS, see the "[Setting Up to Use CROSS PlatformS \(CROPS\)](#)" section in the Yocto Project Development Tasks Manual.

If your development host is going to be a system that runs a Linux distribution, steps still exist that you must take to prepare the system for use with the Yocto Project. You need to be sure that the Linux distribution on the system is one that supports the Yocto Project. You also need to be sure that the correct set of host packages are installed that allow development using the Yocto Project. For the steps needed to set up a development host that runs Linux, see the "[Setting Up a Native Linux Host](#)" section in the Yocto Project Development Tasks Manual.

Once your development host is set up to use the Yocto Project, several methods exist for you to do work in the Yocto Project environment:

- **Command Lines, BitBake, and Shells:** Traditional development in the Yocto Project involves using the [OpenEmbedded build system](#), which uses BitBake, in a command-line environment from a shell on your development host. You can accomplish this from a host that is a native Linux machine or from a host that has been set up with CROPS. Either way, you create, modify, and build images and applications all within a shell-based environment using components and tools available through your Linux distribution and the Yocto Project.

For a general flow of the build procedures, see the "[Building a Simple Image](#)" section in the Yocto Project Development Tasks Manual.

- **Board Support Package (BSP) Development:** Development of BSPs involves using the Yocto Project to create and test layers that allow easy development of images and applications targeted for specific hardware. To development BSPs, you need to take some additional steps beyond what was described in setting up a development host.

The Yocto Project Board Support Package (BSP) Developer's Guide provides BSP-related development information. For specifics on development host preparation, see the "[Preparing Your Build Host to Work With BSP Layers](#)" section in the Yocto Project Board Support Package (BSP) Developer's Guide.

- **Kernel Development:** If you are going to be developing kernels using the Yocto Project you likely will be using `devtool`. A workflow using `devtool` makes kernel development quicker by reducing iteration cycle times.

The Yocto Project Linux Kernel Development Manual provides kernel-related development information. For specifics on development host preparation, see the "[Preparing the Build Host to Work on the Kernel](#)" section in the Yocto Project Linux Kernel Development Manual.

- **Using Toaster:** The other Yocto Project development method that involves an interface that effectively puts the Yocto Project into the background is Toaster. Toaster provides an interface to the OpenEmbedded build system. The interface enables you to configure and run your builds. Information about builds is collected and stored in a database. You can use Toaster to configure and start builds on multiple remote build servers.

For steps that show you how to set up your development host to use Toaster and on how to use Toaster in general, see the Toaster User Manual.

3.3. Yocto Project Source Repositories¶

The Yocto Project team maintains complete source repositories for all Yocto Project files at <http://git.yoctoproject.org>. This web-based source code browser is organized into categories by function such as IDE Plugins, Matchbox, Poky, Yocto Linux Kernel, and so forth. From the interface, you can click on any particular item in the "Name" column and see the URL at the bottom of the page that you need to clone a Git repository for that particular item. Having a local Git repository of the [Source Directory](#), which is usually named "poky", allows you to make changes, contribute to the history, and ultimately enhance the Yocto Project's tools, Board Support Packages, and so forth.

For any supported release of Yocto Project, you can also go to the [Yocto Project Website](#) and select the "DOWNLOADS" item from the "SOFTWARE" menu and get a released tarball of the `poky` repository, any supported BSP tarball, or Yocto Project tools. Unpacking these tarballs gives you a snapshot of the released files.

Notes

- The recommended method for setting up the Yocto Project [Source Directory](#) and the files for supported BSPs (e.g., `meta-intel`) is to use [Git](#) to create a local copy of the upstream repositories.
- Be sure to always work in matching branches for both the selected BSP repository and the Source Directory (i.e. `poky`) repository. For example, if you have checked out the "master" branch of `poky` and you are going to use `meta-intel`, be sure to checkout the "master" branch of `meta-intel`.

In summary, here is where you can get the project files needed for development:

- **Source Repositories:** This area contains IDE Plugins, Matchbox, Poky, Poky Support, Tools, Yocto Linux Kernel, and Yocto Metadata Layers. You can create local copies of Git repositories for each of these areas.

Name	Description	Owner	Idle	Links
poky	Poky Build Tool and Metadata	git repository hosting	1 year	summary log tree
poky-docs	Versioned project documentation	git repository hosting	46 hours	summary log tree
poky-support				
poky-buildhistory	Autobuilder Saved Build History (for Poky master)	git repository hosting	1 year	summary log tree
poky-buildhistory-dev	Autobuilder Saved Build History (for development and testing)	git repository hosting	10 months	summary log tree
poky-buildstats		git repository hosting	5 years	summary log tree
poky-config		git repository hosting	15 months	summary log tree
poky-contrib	Poky Build Tool and Metadata - User Contributions Tree	git repository hosting	1 year	summary log tree
poky-contrib-archive	User contributions older than January 1st 2013	git repository hosting	4 years	summary log tree
yp-qa-build-perf-data	Build performance test results from the Yocto project	git repository hosting	1 year	summary log tree
Yocto Metadata Layers				
intel-iot-refkit	IoT Reference OS Kit for Intel	git repository hosting	1 year	summary log tree
meta-alexa-demo	Layer containing recipes to add an Alexa client to an image	git repository hosting	2 months	summary log tree
meta-amd	Layer containing AMD hardware support metadata	git repository hosting	10 years	summary log tree
meta-axia	Layer for LSI's family of Axia mobile & enterprise communication processors	git repository hosting	7 months	summary log tree
meta-cgl	Enable Carrier Grade Linux compliance through a reference Linux distribution	git repository hosting	1 year	summary log tree
meta-cloud-services	Provides the packages and images for Cloud compute, control and storage nodes (IO..	git repository hosting	14 hours	summary log tree
meta-darwin	Layer for darwin based SDKs	git repository hosting	2 years	summary log tree
meta-darwin-contrib	Darwin based SDKs extras	git repository hosting	2 years	summary log tree
meta-dpdk	Data Plane Developer Kit (dpdk.org)	git repository hosting	3 weeks	summary log tree
meta-external-toolchain	OE/rocks external toolchain support layer	git repository hosting	3 weeks	summary log tree
meta-freescale	Layer containing NXP hardware support metadata	git repository hosting	1 year	summary log tree
meta-gplv2	GPLv2 versions of software where upstream has moved to GPLv3 licenses	git repository hosting	2 weeks	summary log tree
meta-intel	Layer containing Intel hardware support metadata	git repository hosting	10 hours	summary log tree

For steps on how to view and access these upstream Git repositories, see the "Accessing Source Repositories" Section in the Yocto Project Development Tasks Manual.

- **Index of /releases:** This is an index of releases such as Poky, Pseudo, installers for cross-development toolchains, miscellaneous support and all released versions of Yocto Project in the form of images or tarballs. Downloading and extracting these files does not produce a local copy of the Git repository but rather a snapshot of a particular release or image.

Index of /releases/

[../](#)
[ahjuta-plugin-sdk/](#)
[bitbake/](#)
[eglibc/](#)
[exmap-console/](#)
[gnu-config/](#)
[libgconf-bridge/](#)
[libowl-av/](#)
[matchbox/](#)
[media/](#)
[miscsupport/](#)
[opkg/](#)
[oprofileui/](#)
[poky/](#)
[poky-archive-images/](#)
[pseudo/](#)
[psplash/](#)
[rpm5/](#)
[sato/](#)
[uninitative/](#)
[xoo/](#)
[xresponse/](#)
[xrestop/](#)
[yocto/](#)

For steps on how to view and access these files, see the "Accessing Index of Releases" section in the Yocto Project Development Tasks Manual.

- **"DOWNLOADS" page for the Yocto Project Website:**

The Yocto Project website includes a "DOWNLOADS" page accessible through the "SOFTWARE" menu that allows you to download any Yocto Project release, tool, and Board Support Package (BSP) in tarball form. The tarballs are similar to those found in the [Index of /releases/](#) area.

yocto PROJECT SOFTWARE DOCS COMMUNITY LEARN ECOSYSTEM ABOUT JOIN

SOFTWARE : DOWNLOADS

RELEASE YPCORE - ROCKO 24.1 - 2018.01.11

The Yocto Project® build system (BitBake and the OE-Core metadata) is packaged with the reference distro (called Poky). This allows you to try out the whole system. You can create a binary image of Poky as is, or alter the Poky recipes and layers for use in your customized work.

Git is preferred to a tarball download. If you use the former, then code updates can easily be applied to your code base. Using the latter requires you to update your code base yourself, and the documents are slightly older.

The tools and bsp's below are not included. You must download them each separately, and they are specific to the version of software you are using.

RELEASE INFORMATION - YPCORE - ROCKO 24.1

TOOLS

- BUILD APPLIANCE - ROCKO 24.1**
The Build Appliance is a virtual machine which enables you to build and boot a custom embedded Linux system with the Yocto Project release.
- BUILDTOOLS SDK - ROCKO 24.1**
This buildtools SDK contains the needed versions of various programs to build Yocto Project on most distributions. (Environment: Ubuntu 14.04)
- ECLIPSE NEON PLUGIN - ROCKO 24.1**
- ECLIPSE OXYGEN PLUGIN - ROCKO 24.1**

For steps on how to use the "DOWNLOADS" page, see the "Using the Downloads Page" section in the Yocto Project Development Tasks Manual.

3.4. Git Workflows and the Yocto Project¶

Developing using the Yocto Project likely requires the use of [Git](#). Git is a free, open source distributed version control system used as part of many collaborative design environments. This section provides workflow concepts using the Yocto Project and Git. In particular,

the information covers basic practices that describe roles and actions in a collaborative development environment.

Note

If you are familiar with this type of development environment, you might not want to read this section.

The Yocto Project files are maintained using Git in "branches" whose Git histories track every change and whose structures provide branches for all diverging functionality. Although there is no need to use Git, many open source projects do so.

For the Yocto Project, a key individual called the "maintainer" is responsible for the integrity of the "master" branch of a given Git repository. The "master" branch is the "upstream" repository from which final or most recent builds of a project occur. The maintainer is responsible for accepting changes from other developers and for organizing the underlying branch structure to reflect release strategies and so forth.

Note

For information on finding out who is responsible for (maintains) a particular area of code in the Yocto Project, see the ["Submitting a Change to the Yocto Project"](#) section of the Yocto Project Development Tasks Manual.

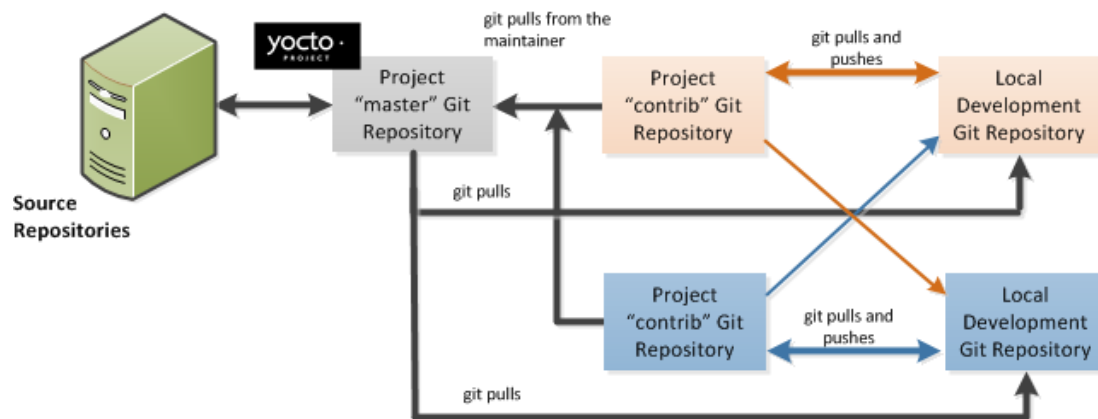
The Yocto Project `poky` Git repository also has an upstream contribution Git repository named `poky-contrib`. You can see all the branches in this repository using the web interface of the [Source Repositories](#) organized within the "Poky Support" area. These branches hold changes (commits) to the project that have been submitted or committed by the Yocto Project development team and by community members who contribute to the project. The maintainer determines if the changes are qualified to be moved from the "contrib" branches into the "master" branch of the Git repository.

Developers (including contributing community members) create and maintain cloned repositories of upstream branches. The cloned repositories are local to their development platforms and are used to develop changes. When a developer is satisfied with a particular feature or change, they "push" the change to the appropriate "contrib" repository.

Developers are responsible for keeping their local repository up-to-date with whatever upstream branch they are working against. They are also responsible for straightening out any conflicts that might arise within files that are being worked on simultaneously by more than one person. All this work is done locally on the development host before anything is pushed to a "contrib" area and examined at the maintainer's level.

A somewhat formal method exists by which developers commit changes and push them into the "contrib" area and subsequently request that the maintainer include them into an upstream branch. This process is called "submitting a patch" or "submitting a change." For information on submitting patches and changes, see the ["Submitting a Change to the Yocto Project"](#) section in the Yocto Project Development Tasks Manual.

In summary, a single point of entry exists for changes into a "master" or development branch of the Git repository, which is controlled by the project's maintainer. And, a set of developers exist who independently develop, test, and submit changes to "contrib" areas for the maintainer to examine. The maintainer then chooses which changes are going to become a permanent part of the project.



While each development environment is unique, there are some best practices or methods that help development run smoothly. The following list describes some of these practices. For more information about Git workflows, see the workflow topics in the [Git Community Book](#).

- **Make Small Changes:** It is best to keep the changes you commit small as compared to bundling many disparate changes into a single commit. This practice not only keeps things manageable but also allows the maintainer to more easily include or refuse changes.
- **Make Complete Changes:** It is also good practice to leave the repository in a state that allows you to still successfully build your project. In other words, do not commit half of a feature, then add the other half as a separate, later commit. Each commit should take you from one buildable project state to another buildable state.
- **Use Branches Liberally:** It is very easy to create, use, and delete local branches in your working Git repository on the development host. You can name these branches anything you like. It is helpful to give them names associated with the particular feature or change on which you are working. Once you are done with a feature or change and have merged it into your local master branch, simply discard the temporary branch.
- **Merge Changes:** The `git merge` command allows you to take the changes from one branch and fold them into another branch. This process is especially helpful when more than a single developer might be working on different parts of the same feature. Merging changes also automatically identifies any collisions or "conflicts" that might happen as a result of the same lines of code being altered by two different developers.

- **Manage Branches:** Because branches are easy to use, you should use a system where branches indicate varying levels of code readiness. For example, you can have a "work" branch to develop in, a "test" branch where the code or change is tested, a "stage" branch where changes are ready to be committed, and so forth. As your project develops, you can merge code across the branches to reflect ever-increasing stable states of the development.
- **Use Push and Pull:** The push-pull workflow is based on the concept of developers "pushing" local commits to a remote repository, which is usually a contribution repository. This workflow is also based on developers "pulling" known states of the project down into their local development repositories. The workflow easily allows you to pull changes submitted by other developers from the upstream repository into your work area ensuring that you have the most recent software on which to develop. The Yocto Project has two scripts named `create-pull-request` and `send-pull-request` that ship with the release to facilitate this workflow. You can find these scripts in the `scripts` folder of the [Source Directory](#). For information on how to use these scripts, see the ["Using Scripts to Push a Change Upstream and Request a Pull"](#) section in the Yocto Project Development Tasks Manual.
- **Patch Workflow:** This workflow allows you to notify the maintainer through an email that you have a change (or patch) you would like considered for the "master" branch of the Git repository. To send this type of change, you format the patch and then send the email using the Git commands `git format-patch` and `git send-email`. For information on how to use these scripts, see the ["Submitting a Change to the Yocto Project"](#) section in the Yocto Project Development Tasks Manual.

3.5. Git¶

The Yocto Project makes extensive use of Git, which is a free, open source distributed version control system. Git supports distributed development, non-linear development, and can handle large projects. It is best that you have some fundamental understanding of how Git tracks projects and how to work with Git if you are going to use the Yocto Project for development. This section provides a quick overview of how Git works and provides you with a summary of some essential Git commands.

Notes

- For more information on Git, see <http://git-scm.com/documentation>.
- If you need to download Git, it is recommended that you add Git to your system through your distribution's "software store" (e.g. for Ubuntu, use the Ubuntu Software feature). For the Git download page, see <http://git-scm.com/download>.
- For information beyond the introductory nature in this section, see the ["Locating Yocto Project Source Files"](#) section in the Yocto Project Development Tasks Manual.

3.5.1. Repositories, Tags, and Branches¶

As mentioned briefly in the previous section and also in the ["Git Workflows and the Yocto Project"](#) section, the Yocto Project maintains source repositories at <http://git.yoctoproject.org>. If you look at this web-interface of the repositories, each item is a separate Git repository.

Git repositories use branching techniques that track content change (not files) within a project (e.g. a new feature or updated documentation). Creating a tree-like structure based on project divergence allows for excellent historical information over the life of a project. This methodology also allows for an environment from which you can do lots of local experimentation on projects as you develop changes or new features.

A Git repository represents all development efforts for a given project. For example, the Git repository `poky` contains all changes and developments for that repository over the course of its entire life. That means that all changes that make up all releases are captured. The repository maintains a complete history of changes.

You can create a local copy of any repository by "cloning" it with the `git clone` command. When you clone a Git repository, you end up with an identical copy of the repository on your development system. Once you have a local copy of a repository, you can take steps to develop locally. For examples on how to clone Git repositories, see the ["Locating Yocto Project Source Files"](#) section in the Yocto Project Development Tasks Manual.

It is important to understand that Git tracks content change and not files. Git uses "branches" to organize different development efforts. For example, the `poky` repository has several branches that include the current "zeus" branch, the "master" branch, and many branches for past Yocto Project releases. You can see all the branches by going to <http://git.yoctoproject.org/cgi.cgi/poky/> and clicking on the [\[branches\]](#) link beneath the "Branch" heading.

Each of these branches represents a specific area of development. The "master" branch represents the current or most recent development. All other branches represent offshoots of the "master" branch.

When you create a local copy of a Git repository, the copy has the same set of branches as the original. This means you can use Git to create a local working area (also called a branch) that tracks a specific development branch from the upstream source Git repository. In other words, you can define your local Git environment to work on any development branch in the repository. To help illustrate, consider the following example Git commands:

```
$ cd ~
$ git clone git://git.yoctoproject.org/poky
$ cd poky
$ git checkout -b zeus origin/zeus
```

In the previous example after moving to the home directory, the `git clone` command creates a local copy of the upstream `poky` Git repository. By default, Git checks out the "master" branch for your work. After changing the working directory to the new local repository (i.e. `poky`), the `git checkout` command creates and checks out a local branch named "zeus", which tracks the upstream "origin/zeus" branch. Changes you make while in this branch would ultimately affect the upstream "zeus" branch of the `poky` repository.

It is important to understand that when you create and checkout a local working branch based on a branch name, your local environment matches the "tip" of that particular development branch at the time you created your local branch, which could be different from the files in the "master" branch of the upstream repository. In other words, creating and checking out a local branch based on the

"zeus" branch name is not the same as checking out the "master" branch in the repository. Keep reading to see how you create a local snapshot of a Yocto Project Release.

Git uses "tags" to mark specific changes in a repository branch structure. Typically, a tag is used to mark a special point such as the final change (or commit) before a project is released. You can see the tags used with the poky Git repository by going to <http://git.yoctoproject.org/cgi.cgi/poky/> and clicking on the `[...]` link beneath the "Tag" heading.

Some key tags for the poky repository are `jethro-14.0.3`, `morty-16.0.1`, `pyro-17.0.0`, and `zeus-22.0.1`. These tags represent Yocto Project releases.

When you create a local copy of the Git repository, you also have access to all the tags in the upstream repository. Similar to branches, you can create and checkout a local working Git branch based on a tag name. When you do this, you get a snapshot of the Git repository that reflects the state of the files when the change was made associated with that tag. The most common use is to checkout a working branch that matches a specific Yocto Project release. Here is an example:

```
$ cd ~
$ git clone git://git.yoctoproject.org/poky
$ cd poky
$ git fetch --tags
$ git checkout tags/rocko-18.0.0 -b my_rocko-18.0.0
```

In this example, the name of the top-level directory of your local Yocto Project repository is `poky`. After moving to the `poky` directory, the `git fetch` command makes all the upstream tags available locally in your repository. Finally, the `git checkout` command creates and checks out a branch named "my-rocko-18.0.0" that is based on the upstream branch whose "HEAD" matches the commit in the repository associated with the "rocko-18.0.0" tag. The files in your repository now exactly match that particular Yocto Project release as it is tagged in the upstream Git repository. It is important to understand that when you create and checkout a local working branch based on a tag, your environment matches a specific point in time and not the entire development branch (i.e. from the "tip" of the branch backwards).

3.5.2. Basic Commands¶

Git has an extensive set of commands that lets you manage changes and perform collaboration over the life of a project. Conveniently though, you can manage with a small set of basic operations and workflows once you understand the basic philosophy behind Git. You do not have to be an expert in Git to be functional. A good place to look for instruction on a minimal set of Git commands is [here](#).

The following list of Git commands briefly describes some basic Git operations as a way to get started. As with any set of commands, this list (in most cases) simply shows the base command and omits the many arguments it supports. See the Git documentation for complete descriptions and strategies on how to use these commands:

- **git init:** Initializes an empty Git repository. You cannot use Git commands unless you have a `.git` repository.
- **git clone:** Creates a local clone of a Git repository that is on equal footing with a fellow developer's Git repository or an upstream repository.
- **git add:** Locally stages updated file contents to the index that Git uses to track changes. You must stage all files that have changed before you can commit them.
- **git commit:** Creates a local "commit" that documents the changes you made. Only changes that have been staged can be committed. Commits are used for historical purposes, for determining if a maintainer of a project will allow the change, and for ultimately pushing the change from your local Git repository into the project's upstream repository.
- **git status:** Reports any modified files that possibly need to be staged and gives you a status of where you stand regarding local commits as compared to the upstream repository.
- **git checkout branch-name:** Changes your local working branch and in this form assumes the local branch already exists. This command is analogous to "cd".
- **git checkout -b working-branch upstream-branch:** Creates and checks out a working branch on your local machine. The local branch tracks the upstream branch. You can use your local branch to isolate your work. It is a good idea to use local branches when adding specific features or changes. Using isolated branches facilitates easy removal of changes if they do not work out.
- **git branch:** Displays the existing local branches associated with your local repository. The branch that you have currently checked out is noted with an asterisk character.
- **git branch -D branch-name:** Deletes an existing local branch. You need to be in a local branch other than the one you are deleting in order to delete *branch-name*.
- **git pull --rebase:** Retrieves information from an upstream Git repository and places it in your local Git repository. You use this command to make sure you are synchronized with the repository from which you are basing changes (e.g. the "master" branch). The "--rebase" option ensures that any local commits you have in your branch are preserved at the top of your local branch.
- **git push repo-name Local-branch : upstream-branch:** Sends all your committed local changes to the upstream Git repository that your local repository is tracking (e.g. a contribution repository). The maintainer of the project draws from these repositories to merge changes (commits) into the appropriate branch of project's upstream repository.
- **git merge:** Combines or adds changes from one local branch of your repository with another branch. When you create a local Git repository, the default branch is named "master". A typical workflow is to create a temporary branch that is based off "master" that you would use for isolated work. You would make your changes in that isolated branch, stage and commit them locally, switch to the "master" branch, and then use the `git merge` command to apply the changes from your isolated branch into the currently checked out branch (e.g. "master"). After the merge is complete and if you are done with working in that isolated branch, you can safely delete the isolated branch.
- **git cherry-pick commits:** Choose and apply specific commits from one branch into another branch. There are times when you might not be able to merge all the changes in one branch with another but need to pick out certain ones.
- **gitk:** Provides a GUI view of the branches and changes in your local Git repository. This command is a good way to graphically see where things have diverged in your local repository.

Note

You need to install the `gitk` package on your development system to use this command.

- **`git log`**: Reports a history of your commits to the repository. This report lists all commits regardless of whether you have pushed them upstream or not.
- **`git diff`**: Displays line-by-line differences between a local working file and the same file as understood by Git. This command is useful to see what you have changed in any given file.

3.6. Licensing¶

Because open source projects are open to the public, they have different licensing structures in place. License evolution for both Open Source and Free Software has an interesting history. If you are interested in this history, you can find basic information here:

- [Open source license history](#).
- [Free software license history](#).

In general, the Yocto Project is broadly licensed under the Massachusetts Institute of Technology (MIT) License. MIT licensing permits the reuse of software within proprietary software as long as the license is distributed with that software. MIT is also compatible with the GNU General Public License (GPL). Patches to the Yocto Project follow the upstream licensing scheme. You can find information on the MIT license [here](#). You can find information on the GNU GPL [here](#).

When you build an image using the Yocto Project, the build process uses a known list of licenses to ensure compliance. You can find this list in the [Source Directory](#) at `meta/files/common-licenses`. Once the build completes, the list of all licenses found and used during that build are kept in the [Build Directory](#) at `tmp/deploy/licenses`.

If a module requires a license that is not in the base list, the build process generates a warning during the build. These tools make it easier for a developer to be certain of the licenses with which their shipped products must comply. However, even with these tools it is still up to the developer to resolve potential licensing issues.

The base list of licenses used by the build process is a combination of the Software Package Data Exchange (SPDX) list and the Open Source Initiative (OSI) projects. [SPDX Group](#) is a working group of the Linux Foundation that maintains a specification for a standard format for communicating the components, licenses, and copyrights associated with a software package. [OSI](#) is a corporation dedicated to the Open Source Definition and the effort for reviewing and approving licenses that conform to the Open Source Definition (OSD).

You can find a list of the combined SPDX and OSI licenses that the Yocto Project uses in the `meta/files/common-licenses` directory in your [Source Directory](#).

For information that can help you maintain compliance with various open source licensing during the lifecycle of a product created using the Yocto Project, see the "Maintaining Open Source License Compliance During Your Product's Lifecycle" section in the Yocto Project Development Tasks Manual.

Chapter 4. Yocto Project Concepts¶

4.1. Yocto Project Components

- [4.1.1. BitBake](#)
- [4.1.2. Recipes](#)
- [4.1.3. Classes](#)
- [4.1.4. Configurations](#)

4.2. Layers

4.3. OpenEmbedded Build System Concepts

- [4.3.1. User Configuration](#)
- [4.3.2. Metadata, Machine Configuration, and Policy Configuration](#)
- [4.3.3. Sources](#)
- [4.3.4. Package Feeds](#)
- [4.3.5. BitBake](#)
- [4.3.6. Images](#)
- [4.3.7. Application Development SDK](#)

4.4. Cross-Development Toolchain Generation

4.5. Shared State Cache

- [4.5.1. Overall Architecture](#)
- [4.5.2. Checksums \(Signatures\)](#)
- [4.5.3. Shared State](#)

4.6. Automatically Added Runtime Dependencies

4.7. Fakeroot and Pseudo

This chapter provides explanations for Yocto Project concepts that go beyond the surface of "how-to" information and reference (or look-up) material. Concepts such as components, the [OpenEmbedded build system](#) workflow, cross-development toolchains, shared state cache, and so forth are explained.

4.1. Yocto Project Components¶

The [BitBake](#) task executor together with various types of configuration files form the [OpenEmbedded-Core](#). This section overviews these components by describing their use and how they interact.

BitBake handles the parsing and execution of the data files. The data itself is of various types:

- **Recipes**: Provides details about particular pieces of software.
- **Class Data**: Abstracts common build information (e.g. how to build a Linux kernel).

- **Configuration Data:** Defines machine-specific settings, policy decisions, and so forth. Configuration data acts as the glue to bind everything together.

BitBake knows how to combine multiple data sources together and refers to each data source as a layer. For information on layers, see the "[Understanding and Creating Layers](#)" section of the Yocto Project Development Tasks Manual.

Following are some brief details on these core components. For additional information on how these components interact during a build, see the "[OpenEmbedded Build System Concepts](#)" section.

4.1.1. BitBake

BitBake is the tool at the heart of the [OpenEmbedded build system](#) and is responsible for parsing the [Metadata](#), generating a list of tasks from it, and then executing those tasks.

This section briefly introduces BitBake. If you want more information on BitBake, see the BitBake User Manual.

To see a list of the options BitBake supports, use either of the following commands:

```
$ bitbake -h
$ bitbake --help
```

The most common usage for BitBake is `bitbake packagename`, where `packagename` is the name of the package you want to build (referred to as the "target"). The target often equates to the first part of a recipe's filename (e.g. "foo" for a recipe named `foo_1.3.0-r0.bb`). So, to process the `matchbox-desktop_1.2.3.bb` recipe file, you might type the following:

```
$ bitbake matchbox-desktop
```

Several different versions of `matchbox-desktop` might exist. BitBake chooses the one selected by the distribution configuration. You can get more details about how BitBake chooses between different target versions and providers in the "[Preferences](#)" section of the BitBake User Manual.

BitBake also tries to execute any dependent tasks first. So for example, before building `matchbox-desktop`, BitBake would build a cross compiler and `glibc` if they had not already been built.

A useful BitBake option to consider is the `-k` or `--continue` option. This option instructs BitBake to try and continue processing the job as long as possible even after encountering an error. When an error occurs, the target that failed and those that depend on it cannot be remade. However, when you use this option other dependencies can still be processed.

4.1.2. Recipes

Files that have the `.bb` suffix are "recipes" files. In general, a recipe contains information about a single piece of software. This information includes the location from which to download the unaltered source, any source patches to be applied to that source (if needed), which special configuration options to apply, how to compile the source files, and how to package the compiled output.

The term "package" is sometimes used to refer to recipes. However, since the word "package" is used for the packaged output from the OpenEmbedded build system (i.e. `.ipk` or `.deb` files), this document avoids using the term "package" when referring to recipes.

4.1.3. Classes

Class files (`.bbclass`) contain information that is useful to share between recipes files. An example is the [autotools](#) class, which contains common settings for any application that Autotools uses. The "[Classes](#)" chapter in the Yocto Project Reference Manual provides details about classes and how to use them.

4.1.4. Configurations

The configuration files (`.conf`) define various configuration variables that govern the OpenEmbedded build process. These files fall into several areas that define machine configuration options, distribution configuration options, compiler tuning options, general common configuration options, and user configuration options in `conf/local.conf`, which is found in the [Build Directory](#).

4.2. Layers

Layers are repositories that contain related metadata (i.e. sets of instructions) that tell the OpenEmbedded build system how to build a target. Yocto Project's [layer model](#) facilitates collaboration, sharing, customization, and reuse within the Yocto Project development environment. Layers logically separate information for your project. For example, you can use a layer to hold all the configurations for a particular piece of hardware. Isolating hardware-specific configurations allows you to share other metadata by using a different layer where that metadata might be common across several pieces of hardware.

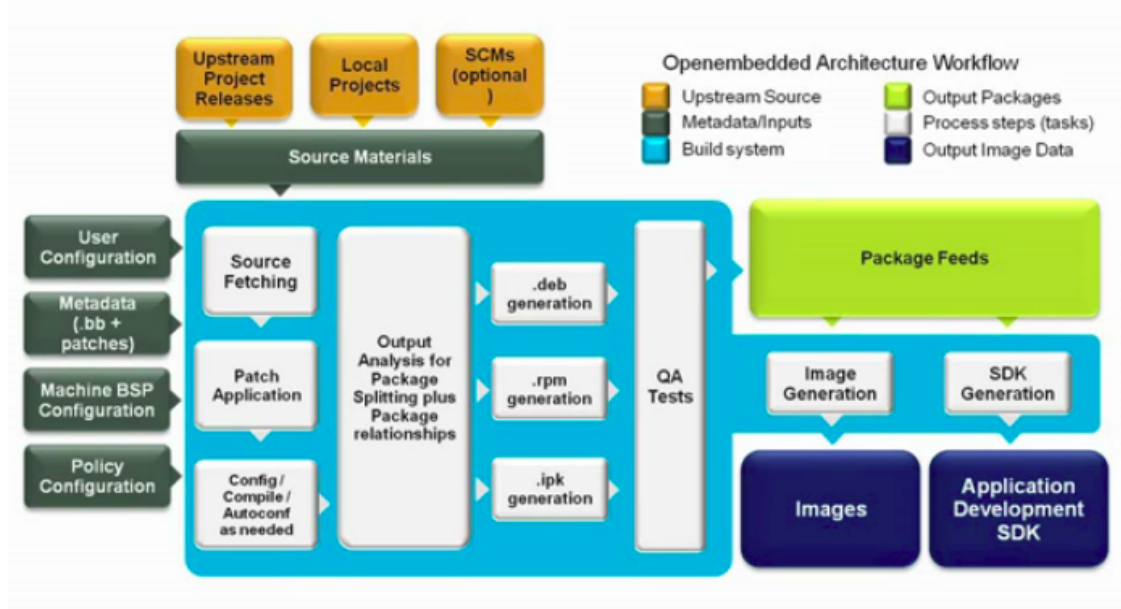
Many layers exist that work in the Yocto Project development environment. The [Yocto Project Curated Layer Index](#) and [OpenEmbedded Layer Index](#) both contain layers from which you can use or leverage.

By convention, layers in the Yocto Project follow a specific form. Conforming to a known structure allows BitBake to make assumptions during builds on where to find types of metadata. You can find procedures and learn about tools (i.e. `bitbake-layers`) for creating layers suitable for the Yocto Project in the "[Understanding and Creating Layers](#)" section of the Yocto Project Development Tasks Manual.

4.3. OpenEmbedded Build System Concepts

This section takes a more detailed look inside the build process used by the [OpenEmbedded build system](#), which is the build system specific to the Yocto Project. At the heart of the build system is BitBake, the task executor.

The following diagram represents the high-level workflow of a build. The remainder of this section expands on the fundamental input, output, process, and metadata logical blocks that make up the workflow.



In general, the build's workflow consists of several functional areas:

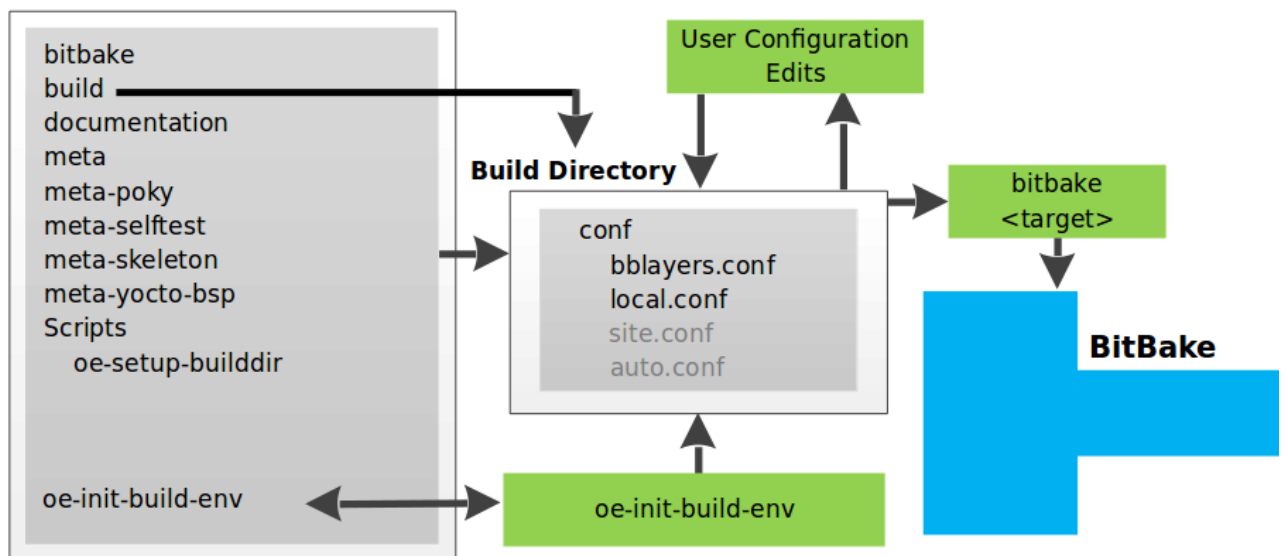
- **User Configuration:** metadata you can use to control the build process.
- **Metadata Layers:** Various layers that provide software, machine, and distro metadata.
- **Source Files:** Upstream releases, local projects, and SCMs.
- **Build System:** Processes under the control of BitBake. This block expands on how BitBake fetches source, applies patches, completes compilation, analyzes output for package generation, creates and tests packages, generates images, and generates cross-development tools.
- **Package Feeds:** Directories containing output packages (RPM, DEB or IPK), which are subsequently used in the construction of an image or Software Development Kit (SDK), produced by the build system. These feeds can also be copied and shared using a web server or other means to facilitate extending or updating existing images on devices at runtime if runtime package management is enabled.
- **Images:** Images produced by the workflow.
- **Application Development SDK:** Cross-development tools that are produced along with an image or separately with BitBake.

4.3.1. User Configuration¶

User configuration helps define the build. Through user configuration, you can tell BitBake the target architecture for which you are building the image, where to store downloaded source, and other build properties.

The following figure shows an expanded representation of the "User Configuration" box of the [general workflow figure](#):

Source Directory (e.g. poky directory)



BitBake needs some basic configuration files in order to complete a build. These files are *.conf files. The minimally necessary ones reside as example files in the build/conf directory of the [Source Directory](#). For simplicity, this section refers to the Source Directory as the "Poky Directory."

When you clone the [Poky](#) Git repository or you download and unpack a Yocto Project release, you can set up the Source Directory to be named anything you want. For this discussion, the cloned repository uses the default name poky.

Note

The Poky repository is primarily an aggregation of existing repositories. It is not a canonical upstream source.

The `meta-poky` layer inside Poky contains a `conf` directory that has example configuration files. These example files are used as a basis for creating actual configuration files when you source `oe-init-build-env`, which is the build environment script.

Sourcing the build environment script creates a `Build Directory` if one does not already exist. BitBake uses the Build Directory for all its work during builds. The Build Directory has a `conf` directory that contains default versions of your `local.conf` and `bblayers.conf` configuration files. These default configuration files are created only if versions do not already exist in the Build Directory at the time you source the build environment setup script.

Because the Poky repository is fundamentally an aggregation of existing repositories, some users might be familiar with running the `oe-init-build-env` script in the context of separate `OpenEmbedded-Core` and BitBake repositories rather than a single Poky repository. This discussion assumes the script is executed from within a cloned or unpacked version of Poky.

Depending on where the script is sourced, different sub-scripts are called to set up the Build Directory (Yocto or OpenEmbedded). Specifically, the script `scripts/oe-setup-builddir` inside the poky directory sets up the Build Directory and seeds the directory (if necessary) with configuration files appropriate for the Yocto Project development environment.

Note

The `scripts/oe-setup-builddir` script uses the `$TEMPLATECONF` variable to determine which sample configuration files to locate.

The `local.conf` file provides many basic variables that define a build environment. Here is a list of a few. To see the default configurations in a `local.conf` file created by the build environment script, see the `local.conf.sample` in the `meta-poky` layer:

- **Target Machine Selection:** Controlled by the `MACHINE` variable.
- **Download Directory:** Controlled by the `DL_DIR` variable.
- **Shared State Directory:** Controlled by the `SSTATE_DIR` variable.
- **Build Output:** Controlled by the `TMPDIR` variable.
- **Distribution Policy:** Controlled by the `DISTRO` variable.
- **Packaging Format:** Controlled by the `PACKAGE_CLASSES` variable.
- **SDK Target Architecture:** Controlled by the `SDKMACHINE` variable.
- **Extra Image Packages:** Controlled by the `EXTRA_IMAGE_FEATURES` variable.

Note

Configurations set in the `conf/local.conf` file can also be set in the `conf/site.conf` and `conf/auto.conf` configuration files.

The `bblayers.conf` file tells BitBake what layers you want considered during the build. By default, the layers listed in this file include layers minimally needed by the build system. However, you must manually add any custom layers you have created. You can find more information on working with the `bblayers.conf` file in the "Enabling Your Layer" section in the Yocto Project Development Tasks Manual.

The files `site.conf` and `auto.conf` are not created by the environment initialization script. If you want the `site.conf` file, you need to create that yourself. The `auto.conf` file is typically created by an autobuilder:

- **site.conf:** You can use the `conf/site.conf` configuration file to configure multiple build directories. For example, suppose you had several build environments and they shared some common features. You can set these default build properties here. A good example is perhaps the packaging format to use through the `PACKAGE_CLASSES` variable.

One useful scenario for using the `conf/site.conf` file is to extend your `BBPATH` variable to include the path to a `conf/site.conf`. Then, when BitBake looks for Metadata using `BBPATH`, it finds the `conf/site.conf` file and applies your common configurations found in the file. To override configurations in a particular build directory, alter the similar configurations within that build directory's `conf/local.conf` file.

- **auto.conf:** The file is usually created and written to by an autobuilder. The settings put into the file are typically the same as you would find in the `conf/local.conf` or the `conf/site.conf` files.

You can edit all configuration files to further define any particular build environment. This process is represented by the "User Configuration Edits" box in the figure.

When you launch your build with the `bitbake target` command, BitBake sorts out the configurations to ultimately define your build environment. It is important to understand that the `OpenEmbedded build system` reads the configuration files in a specific order: `site.conf`, `auto.conf`, and `local.conf`. And, the build system applies the normal assignment statement rules as described in the "Syntax and Operators" chapter of the BitBake User Manual. Because the files are parsed in a specific order, variable assignments for the same variable could be affected. For example, if the `auto.conf` file and the `local.conf` set `variable1` to different values, because the build system parses `local.conf` after `auto.conf`, `variable1` is assigned the value from the `local.conf` file.

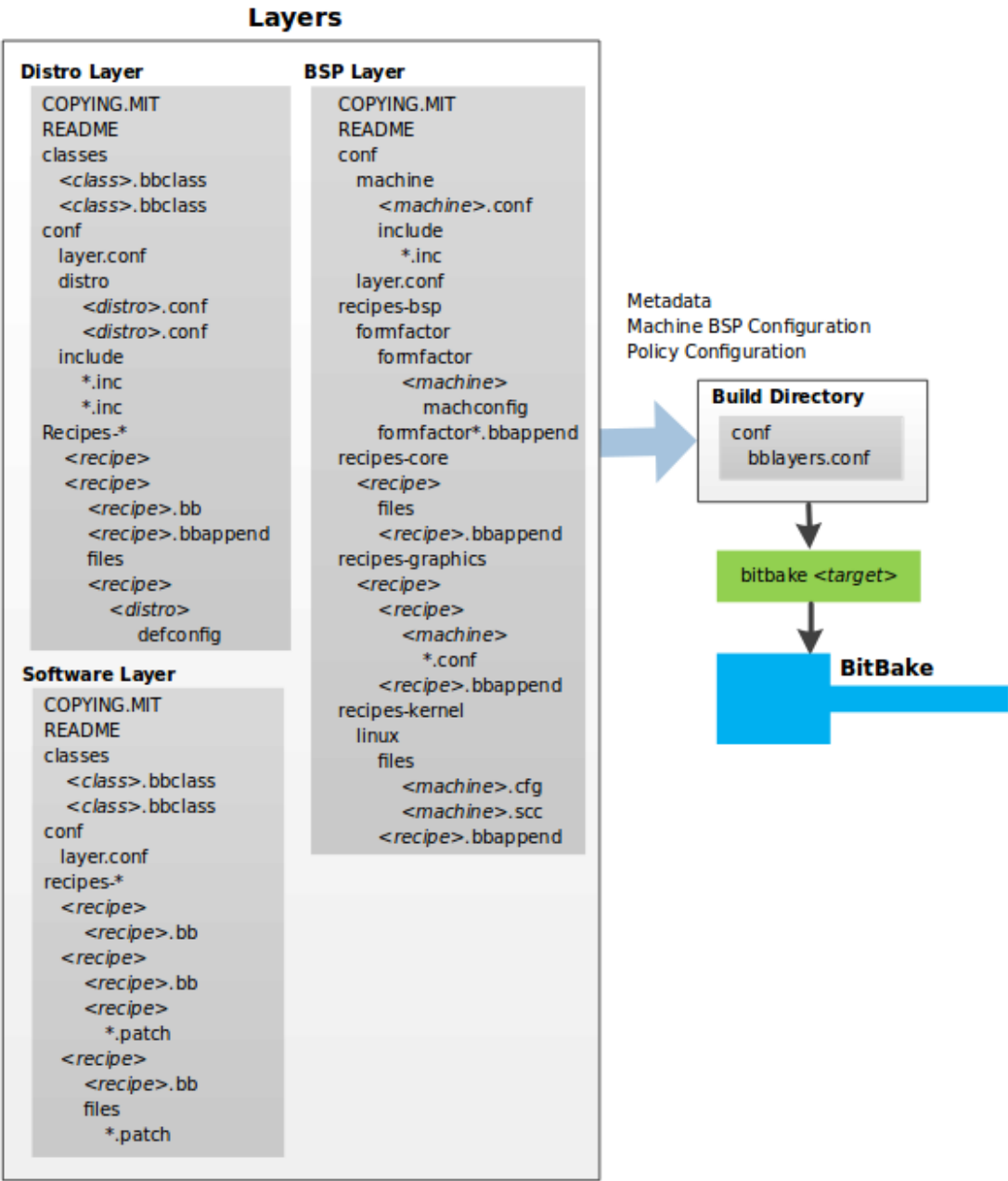
4.3.2. Metadata, Machine Configuration, and Policy Configuration¶

The previous section described the user configurations that define BitBake's global behavior. This section takes a closer look at the layers the build system uses to further control the build. These layers provide Metadata for the software, machine, and policies.

In general, three types of layer input exists. You can see them below the "User Configuration" box in the [general workflow figure](#):

- **Metadata (.bb + Patches):** Software layers containing user-supplied recipe files, patches, and append files. A good example of a software layer might be the `meta-qt5` layer from the [OpenEmbedded Layer Index](#). This layer is for version 5.0 of the popular Qt cross-platform application development framework for desktop, embedded and mobile.
- **Machine BSP Configuration:** Board Support Package (BSP) layers (i.e. "BSP Layer" in the following figure) providing machine-specific configurations. This type of information is specific to a particular target architecture. A good example of a BSP layer from the Poky Reference Distribution is the `meta-yocto-bsp` layer.
- **Policy Configuration:** Distribution Layers (i.e. "Distro Layer" in the following figure) providing top-level or general policies for the images or SDKs being built for a particular distribution. For example, in the Poky Reference Distribution the distro layer is the `meta-poky` layer. Within the distro layer is a `conf/distro` directory that contains distro configuration files (e.g. `poky.conf`) that contain many policy configurations for the Poky distribution.

The following figure shows an expanded representation of these three layers from the [general workflow figure](#):



In general, all layers have a similar structure. They all contain a licensing file (e.g. `COPYING.MIT`) if the layer is to be distributed, a `README` file as good practice and especially if the layer is to be distributed, a configuration directory, and recipe directories. You can learn about the general structure for layers used with the Yocto Project in the "Creating Your Own Layer" section in the Yocto Project Development Tasks Manual. For a general discussion on layers and the many layers from which you can draw, see the "Layers" and "The Yocto Project Layer Model" sections both earlier in this manual.

If you explored the previous links, you discovered some areas where many layers that work with the Yocto Project exist. The [Source Repositories](#) also shows layers categorized under "Yocto Metadata Layers."

Note

Layers exist in the Yocto Project Source Repositories that cannot be found in the OpenEmbedded Layer Index. These layers are either deprecated or experimental in nature.

BitBake uses the `conf/bblayers.conf` file, which is part of the user configuration, to find what layers it should be using as part of the build.

4.3.2.1. Distro Layer

The distribution layer provides policy configurations for your distribution. Best practices dictate that you isolate these types of configurations into their own layer. Settings you provide in `conf/distro/distro.conf` override similar settings that BitBake finds in your `conf/local.conf` file in the Build Directory.

The following list provides some explanation and references for what you typically find in the distribution layer:

- **classes:** Class files (`.bbclass`) hold common functionality that can be shared among recipes in the distribution. When your recipes inherit a class, they take on the settings and functions for that class. You can read more about class files in the ["Classes"](#) chapter of the Yocto Reference Manual.
- **conf:** This area holds configuration files for the layer (`conf/layer.conf`), the distribution (`conf/distro/distro.conf`), and any distribution-wide include files.
- **recipes-***: Recipes and append files that affect common functionality across the distribution. This area could include recipes and append files to add distribution-specific configuration, initialization scripts, custom image recipes, and so forth. Examples of `recipes-*` directories are `recipes-core` and `recipes-extra`. Hierarchy and contents within a `recipes-*` directory can vary. Generally, these directories contain recipe files (`*.bb`), recipe append files (`*.bbappend`), directories that are distro-specific for configuration files, and so forth.

4.3.2.2. BSP Layer

The BSP Layer provides machine configurations that target specific hardware. Everything in this layer is specific to the machine for which you are building the image or the SDK. A common structure or form is defined for BSP layers. You can learn more about this structure in the Yocto Project Board Support Package (BSP) Developer's Guide.

Note

In order for a BSP layer to be considered compliant with the Yocto Project, it must meet some structural requirements.

The BSP Layer's configuration directory contains configuration files for the machine (`conf/machine/machine.conf`) and, of course, the layer (`conf/layer.conf`).

The remainder of the layer is dedicated to specific recipes by function: `recipes-bsp`, `recipes-core`, `recipes-graphics`, `recipes-kernel`, and so forth. Metadata can exist for multiple formfactors, graphics support systems, and so forth.

Note

While the figure shows several `recipes-*` directories, not all these directories appear in all BSP layers.

4.3.2.3. Software Layer

The software layer provides the Metadata for additional software packages used during the build. This layer does not include Metadata that is specific to the distribution or the machine, which are found in their respective layers.

This layer contains any recipes, append files, and patches, that your project needs.

4.3.3. Sources

In order for the OpenEmbedded build system to create an image or any target, it must be able to access source files. The [general workflow figure](#) represents source files using the "Upstream Project Releases", "Local Projects", and "SCMs (optional)" boxes. The figure represents mirrors, which also play a role in locating source files, with the "Source Materials" box.

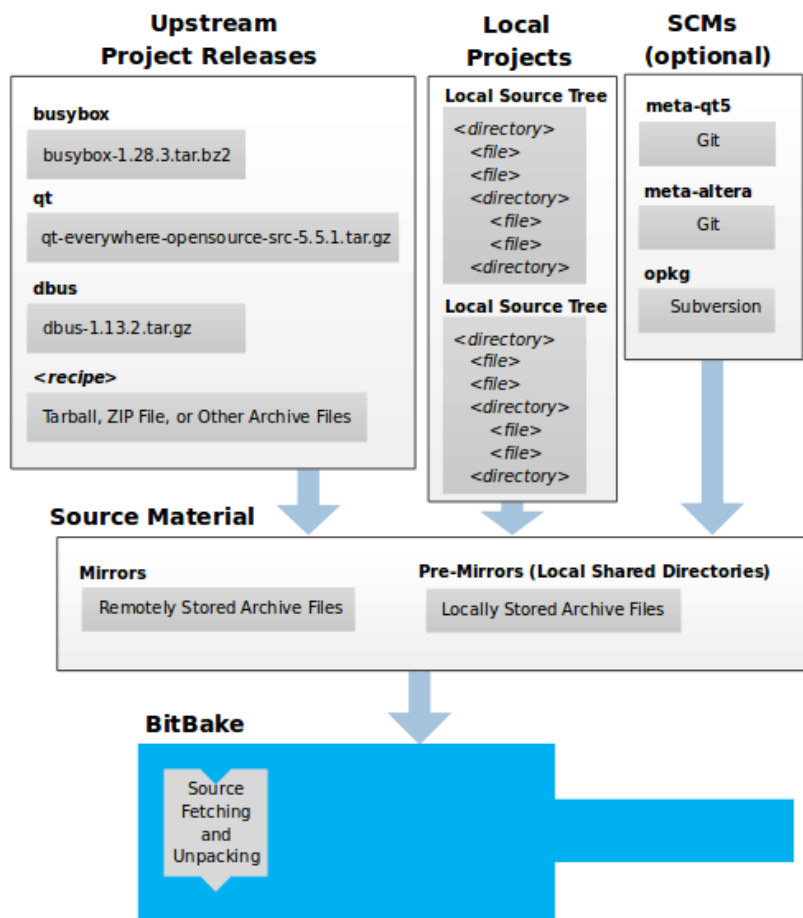
The method by which source files are ultimately organized is a function of the project. For example, for released software, projects tend to use tarballs or other archived files that can capture the state of a release guaranteeing that it is statically represented. On the other hand, for a project that is more dynamic or experimental in nature, a project might keep source files in a repository controlled by a Source Control Manager (SCM) such as Git. Pulling source from a repository allows you to control the point in the repository (the revision) from which you want to build software. Finally, a combination of the two might exist, which would give the consumer a choice when deciding where to get source files.

BitBake uses the `SRC_URI` variable to point to source files regardless of their location. Each recipe must have a `SRC_URI` variable that points to the source.

Another area that plays a significant role in where source files come from is pointed to by the `DL_DIR` variable. This area is a cache that can hold previously downloaded source. You can also instruct the OpenEmbedded build system to create tarballs from Git repositories, which is not the default behavior, and store them in the `DL_DIR` by using the `BB_GENERATE_MIRROR_TARBALLS` variable.

Judicious use of a `DL_DIR` directory can save the build system a trip across the Internet when looking for files. A good method for using a download directory is to have `DL_DIR` point to an area outside of your Build Directory. Doing so allows you to safely delete the Build Directory if needed without fear of removing any downloaded source file.

The remainder of this section provides a deeper look into the source files and the mirrors. Here is a more detailed look at the source file area of the [general workflow figure](#):



4.3.3.1. Upstream Project Releases¶

Upstream project releases exist anywhere in the form of an archived file (e.g. tarball or zip file). These files correspond to individual recipes. For example, the figure uses specific releases each for BusyBox, Qt, and Dbus. An archive file can be for any released product that can be built using a recipe.

4.3.3.2. Local Projects¶

Local projects are custom bits of software the user provides. These bits reside somewhere local to a project - perhaps a directory into which the user checks in items (e.g. a local directory containing a development source tree used by the group).

The canonical method through which to include a local project is to use the `externalsrc` class to include that local project. You use either the `local.conf` or a recipe's append file to override or set the recipe to point to the local directory on your disk to pull in the whole source tree.

4.3.3.3. Source Control Managers (Optional)¶

Another place from which the build system can get source files is with `fetchers` employing various Source Control Managers (SCMs) such as Git or Subversion. In such cases, a repository is cloned or checked out. The `do_fetch` task inside BitBake uses the `SRC_URI` variable and the argument's prefix to determine the correct fetcher module.

Note

For information on how to have the OpenEmbedded build system generate tarballs for Git repositories and place them in the `DL_DIR` directory, see the `BB_GENERATE_MIRROR_TARBALLS` variable in the Yocto Project Reference Manual.

When fetching a repository, BitBake uses the `SRCREV` variable to determine the specific revision from which to build.

4.3.3.4. Source Mirror(s)¶

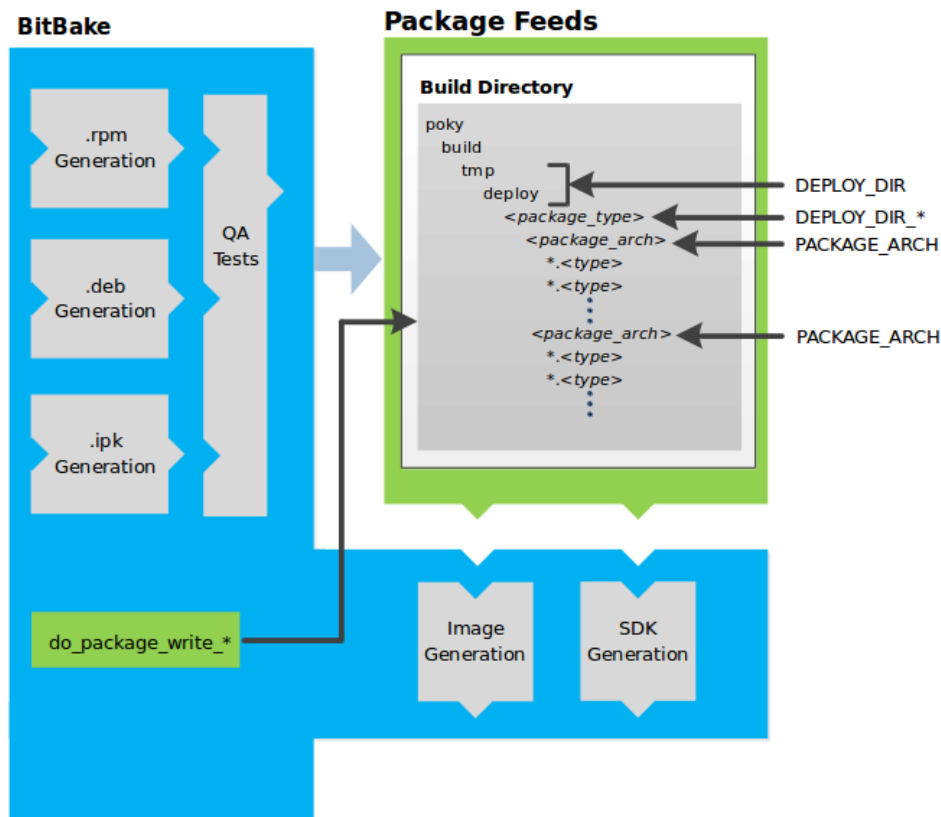
Two kinds of mirrors exist: pre-mirrors and regular mirrors. The `PREMIRRORS` and `MIRRORS` variables point to these, respectively. BitBake checks pre-mirrors before looking upstream for any source files. Pre-mirrors are appropriate when you have a shared directory that is not a directory defined by the `DL_DIR` variable. A Pre-mirror typically points to a shared directory that is local to your organization.

Regular mirrors can be any site across the Internet that is used as an alternative location for source code should the primary site not be functioning for some reason or another.

4.3.4. Package Feeds

When the OpenEmbedded build system generates an image or an SDK, it gets the packages from a package feed area located in the [Build Directory](#). The [general workflow figure](#) shows this package feeds area in the upper-right corner.

This section looks a little closer into the package feeds area used by the build system. Here is a more detailed look at the area:



Package feeds are an intermediary step in the build process. The OpenEmbedded build system provides classes to generate different package types, and you specify which classes to enable through the [PACKAGE_CLASSES](#) variable. Before placing the packages into package feeds, the build process validates them with generated output quality assurance checks through the [insane](#) class.

The package feed area resides in the Build Directory. The directory the build system uses to temporarily store packages is determined by a combination of variables and the particular package manager in use. See the "Package Feeds" box in the illustration and note the information to the right of that area. In particular, the following defines where package files are kept:

- [DEPLOY_DIR](#): Defined as `tmp/deploy` in the Build Directory.
- [DEPLOY_DIR_*](#): Depending on the package manager used, the package type sub-folder. Given RPM, IPK, or DEB packaging and tarball creation, the [DEPLOY_DIR_RPM](#), [DEPLOY_DIR_IPK](#), [DEPLOY_DIR_DEB](#), or [DEPLOY_DIR_TAR](#), variables are used, respectively.
- [PACKAGE_ARCH](#): Defines architecture-specific sub-folders. For example, packages could exist for the i586 or qemux86 architectures.

BitBake uses the `do_package_write_*` tasks to generate packages and place them into the package holding area (e.g. `do_package_write_ipk` for IPK packages). See the "[do_package_write_deb](#)", "[do_package_write_ipk](#)", "[do_package_write_rpm](#)", and "[do_package_write_tar](#)" sections in the Yocto Project Reference Manual for additional information. As an example, consider a scenario where an IPK packaging manager is being used and package architecture support for both i586 and qemux86 exist. Packages for the i586 architecture are placed in `build/tmp/deploy/ipk/i586`, while packages for the qemux86 architecture are placed in `build/tmp/deploy/ipk/qemux86`.

4.3.5. BitBake

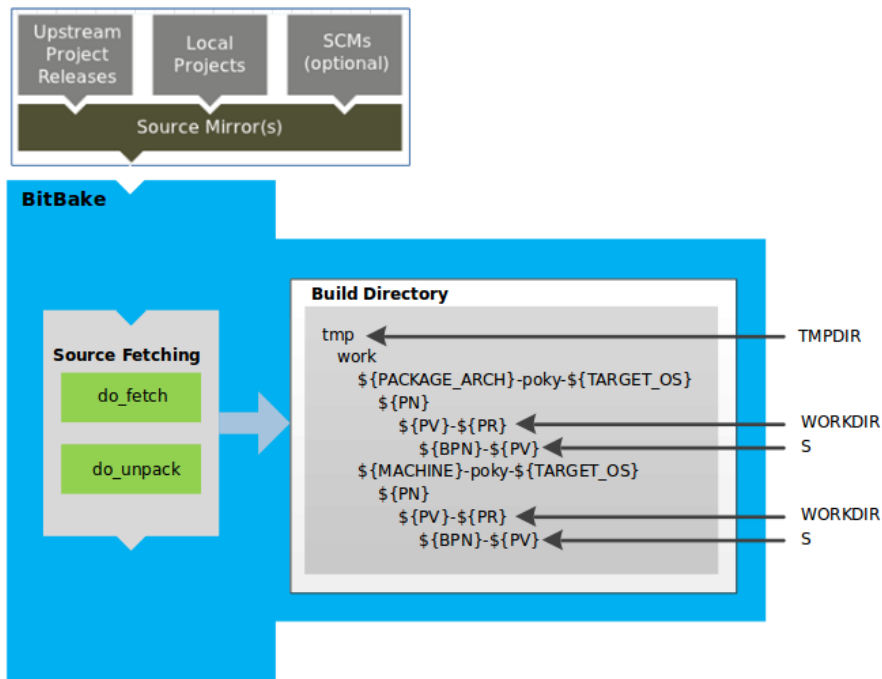
The OpenEmbedded build system uses [BitBake](#) to produce images and Software Development Kits (SDKs). You can see from the [general workflow figure](#), the BitBake area consists of several functional areas. This section takes a closer look at each of those areas.

Note

Separate documentation exists for the BitBake tool. See the BitBake User Manual for reference material on BitBake.

4.3.5.1. Source Fetching

The first stages of building a recipe are to fetch and unpack the source code:



The `do_fetch` and `do_unpack` tasks fetch the source files and unpack them into the `Build Directory`.

Note

For every local file (e.g. `file://`) that is part of a recipe's `SRC_URI` statement, the OpenEmbedded build system takes a checksum of the file for the recipe and inserts the checksum into the signature for the `do_fetch` task. If any local file has been modified, the `do_fetch` task and all tasks that depend on it are re-executed.

By default, everything is accomplished in the Build Directory, which has a defined structure. For additional general information on the Build Directory, see the "[build/](#)" section in the Yocto Project Reference Manual.

Each recipe has an area in the Build Directory where the unpacked source code resides. The `S` variable points to this area for a recipe's unpacked source code. The name of that directory for any given recipe is defined from several different variables. The preceding figure and the following list describe the Build Directory's hierarchy:

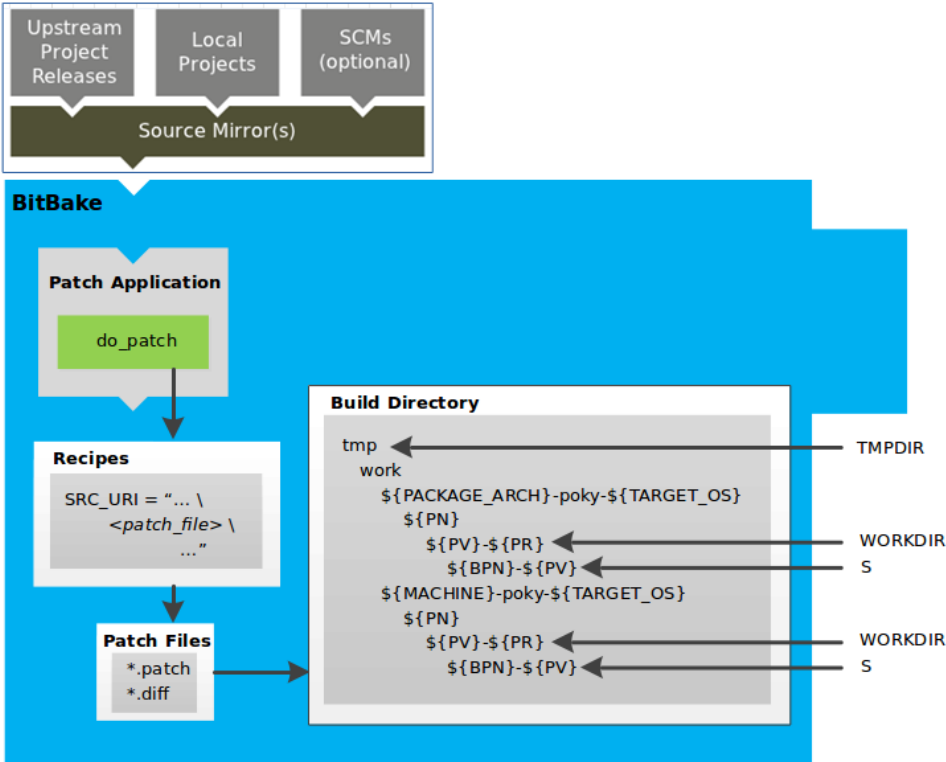
- **`TMPDIR`**: The base directory where the OpenEmbedded build system performs all its work during the build. The default base directory is the `tmp` directory.
- **`PACKAGE_ARCH`**: The architecture of the built package or packages. Depending on the eventual destination of the package or packages (i.e. machine architecture, `build host`, SDK, or specific machine), `PACKAGE_ARCH` varies. See the variable's description for details.
- **`TARGET_OS`**: The operating system of the target device. A typical value would be "linux" (e.g. "qemux86-poky-linux").
- **`PN`**: The name of the recipe used to build the package. This variable can have multiple meanings. However, when used in the context of input files, `PN` represents the name of the recipe.
- **`WORKDIR`**: The location where the OpenEmbedded build system builds a recipe (i.e. does the work to create the package).
 - **`PV`**: The version of the recipe used to build the package.
 - **`PR`**: The revision of the recipe used to build the package.
- **`S`**: Contains the unpacked source files for a given recipe.
 - **`BPN`**: The name of the recipe used to build the package. The `BPN` variable is a version of the `PN` variable but with common prefixes and suffixes removed.
 - **`PV`**: The version of the recipe used to build the package.

Note

In the previous figure, notice that two sample hierarchies exist: one based on package architecture (i.e. `PACKAGE_ARCH`) and one based on a machine (i.e. `MACHINE`). The underlying structures are identical. The differentiator being what the OpenEmbedded build system is using as a build target (e.g. general architecture, a build host, an SDK, or a specific machine).

4.3.5.2. Patching¶

Once source code is fetched and unpacked, BitBake locates patch files and applies them to the source files:



The `do_patch` task uses a recipe's `SRC_URI` statements and the `FILES_PATH` variable to locate applicable patch files.

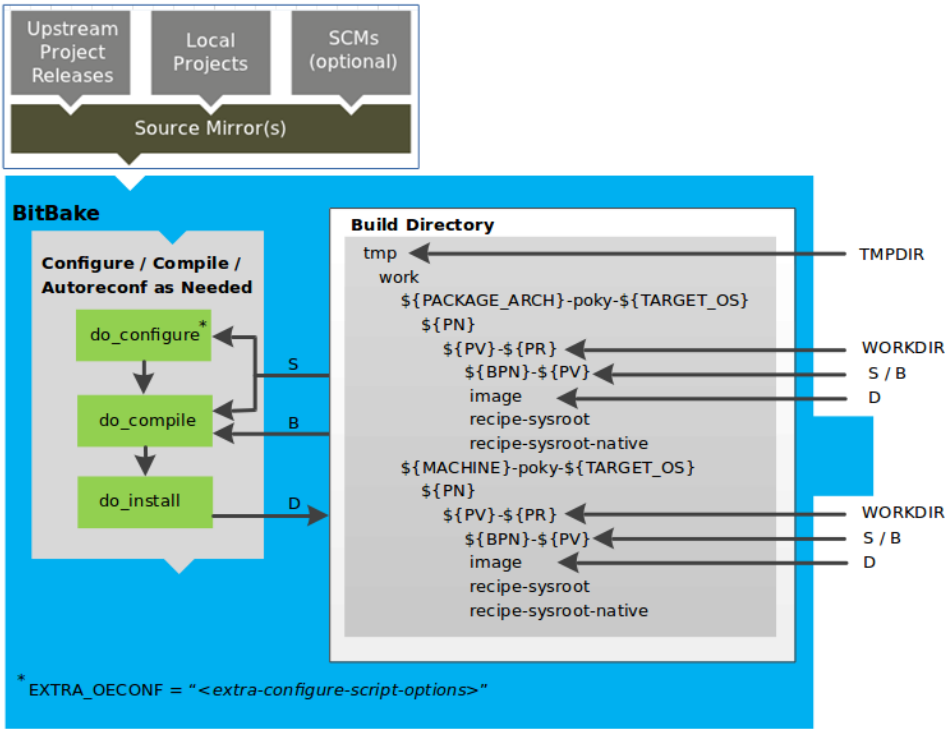
Default processing for patch files assumes the files have either `*.patch` or `*.diff` file types. You can use `SRC_URI` parameters to change the way the build system recognizes patch files. See the `do_patch` task for more information.

BitBake finds and applies multiple patches for a single recipe in the order in which it locates the patches. The `FILES_PATH` variable defines the default set of directories that the build system uses to search for patch files. Once found, patches are applied to the recipe's source files, which are located in the `S` directory.

For more information on how the source directories are created, see the "Source Fetching" section. For more information on how to create patches and how the build system processes patches, see the "Patching Code" section in the Yocto Project Development Tasks Manual. You can also see the "Use `devtool modify` to Modify the Source of an Existing Component" section in the Yocto Project Application Development and the Extensible Software Development Kit (SDK) manual and the "Using Traditional Kernel Development to Patch the Kernel" section in the Yocto Project Linux Kernel Development Manual.

4.3.5.3. Configuration, Compilation, and Staging¶

After source code is patched, BitBake executes tasks that configure and compile the source code. Once compilation occurs, the files are copied to a holding area (staged) in preparation for packaging:



This step in the build process consists of the following tasks:

- **do_prepare_recipe_sysroot**: This task sets up the two sysroots in `${WORKDIR}` (i.e. `recipe-sysroot` and `recipe-sysroot-native`) so that during the packaging phase the sysroots can contain the contents of the `do_populate_sysroot` tasks of the recipes on which the recipe containing the tasks depends. A sysroot exists for both the target and for the native binaries, which run on the host system.
- **do_configure**: This task configures the source by enabling and disabling any build-time and configuration options for the software being built. Configurations can come from the recipe itself as well as from an inherited class. Additionally, the software itself might configure itself depending on the target for which it is being built.

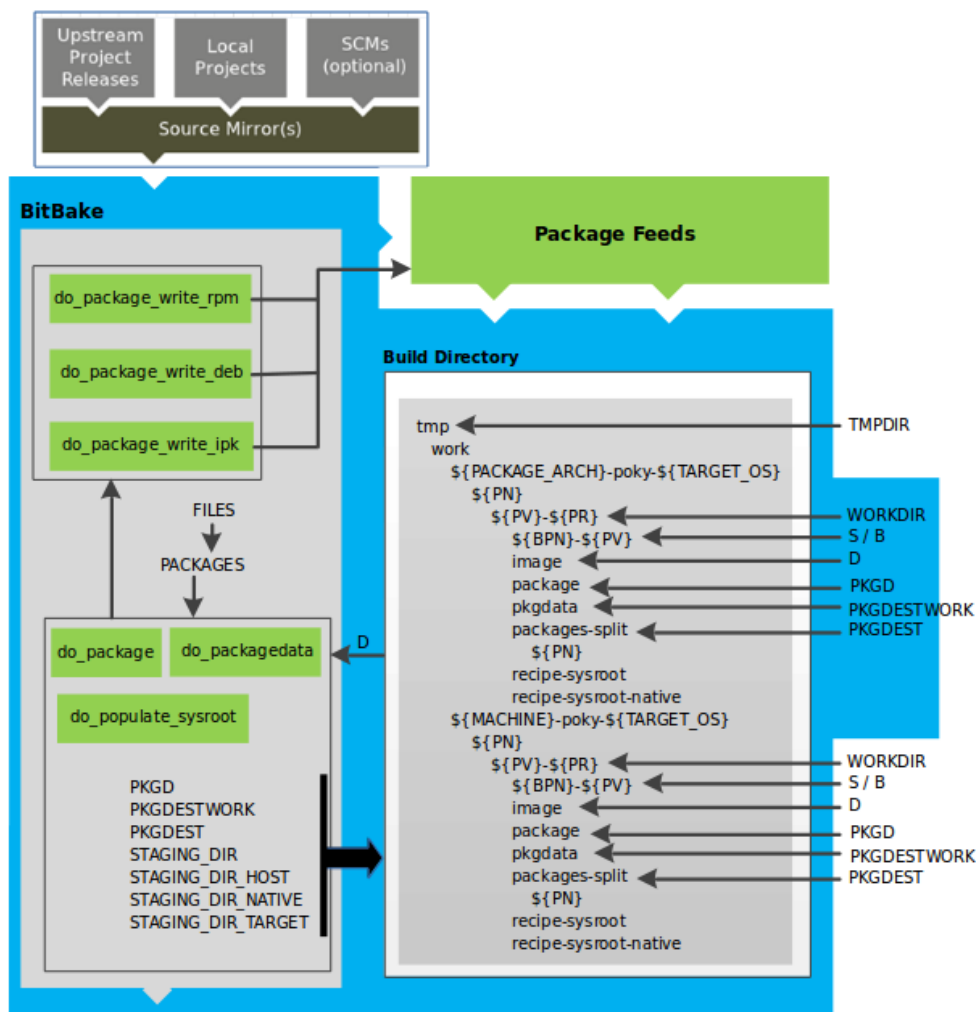
The configurations handled by the `do_configure` task are specific to configurations for the source code being built by the recipe.

If you are using the `autotools` class, you can add additional configuration options by using the `EXTRA_OECONF` or `PACKAGECONFIG_CONFARGS` variables. For information on how this variable works within that class, see the `autotools` class [here](#).

- **do_compile**: Once a configuration task has been satisfied, BitBake compiles the source using the `do_compile` task. Compilation occurs in the directory pointed to by the `B` variable. Realize that the `B` directory is, by default, the same as the `S` directory.
- **do_install**: After compilation completes, BitBake executes the `do_install` task. This task copies files from the `B` directory and places them in a holding area pointed to by the `D` variable. Packaging occurs later using files from this holding directory.

4.3.5.4. Package Splitting¶

After source code is configured, compiled, and staged, the build system analyzes the results and splits the output into packages:



The `do_package` and `do_packagedata` tasks combine to analyze the files found in the `D` directory and split them into subsets based on available packages and files. Analysis involves the following as well as other items: splitting out debugging symbols, looking at shared library dependencies between packages, and looking at package relationships.

The `do_packagedata` task creates package metadata based on the analysis such that the build system can generate the final packages. The `do_populate_sysroot` task stages (copies) a subset of the files installed by the `do_install` task into the appropriate sysroot. Working, staged, and intermediate results of the analysis and package splitting process use several areas:

- **PKGDEST**: The destination directory (i.e. `package`) for packages before they are split into individual packages.
- **PKGDESTWORK**: A temporary work area (i.e. `pkgdata`) used by the `do_package` task to save package metadata.
- **PKGDEST**: The parent directory (i.e. `packages-split`) for packages after they have been split.
- **PKGDATA_DIR**: A shared, global-state directory that holds packaging metadata generated during the packaging process. The packaging process copies metadata from `PKGDESTWORK` to the `PKGDATA_DIR` area where it becomes globally available.

- STAGING_DIR_HOST: The path for the sysroot for the system on which a component is built to run (i.e. `recipe-sysroot`).
- STAGING_DIR_NATIVE: The path for the sysroot used when building components for the build host (i.e. `recipe-sysroot-native`).
- STAGING_DIR_TARGET: The path for the sysroot used when a component that is built to execute on a system and it generates code for yet another machine (e.g. `cross-canadian-recipes`).

The FILES variable defines the files that go into each package in PACKAGES. If you want details on how this is accomplished, you can look at package.bbclass.

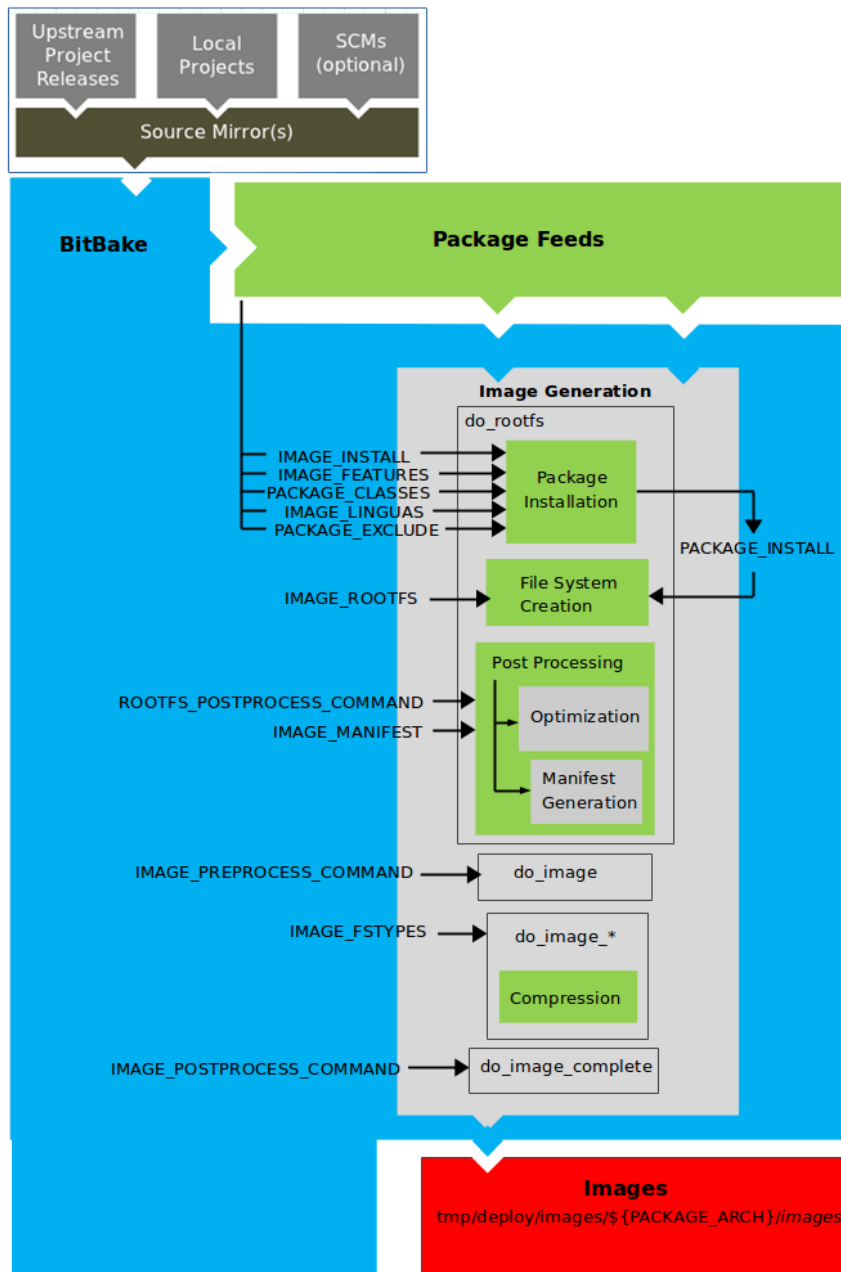
Depending on the type of packages being created (RPM, DEB, or IPK), the do_package_write * task creates the actual packages and places them in the Package Feed area, which is `${TMPDIR}/deploy`. You can see the "Package Feeds" section for more detail on that part of the build process.

Note

Support for creating feeds directly from the `deploy/*` directories does not exist. Creating such feeds usually requires some kind of feed maintenance mechanism that would upload the new packages into an official package feed (e.g. the Ångström distribution). This functionality is highly distribution-specific and thus is not provided out of the box.

4.3.5.5. Image Generation

Once packages are split and stored in the Package Feeds area, the build system uses BitBake to generate the root filesystem image:



The image generation process consists of several stages and depends on several tasks and variables. The `do_rootfs` task creates the root filesystem (file and directory structure) for an image. This task uses several key variables to help create the list of packages to actually install:

- `IMAGE_INSTALL`: Lists out the base set of packages from which to install from the Package Feeds area.
- `PACKAGE_EXCLUDE`: Specifies packages that should not be installed into the image.
- `IMAGE_FEATURES`: Specifies features to include in the image. Most of these features map to additional packages for installation.
- `PACKAGE_CLASSES`: Specifies the package backend (e.g. RPM, DEB, or IPK) to use and consequently helps determine where to locate packages within the Package Feeds area.
- `IMAGE_LINGUAS`: Determines the language(s) for which additional language support packages are installed.
- `PACKAGE_INSTALL`: The final list of packages passed to the package manager for installation into the image.

With `IMAGE_ROOTFS` pointing to the location of the filesystem under construction and the `PACKAGE_INSTALL` variable providing the final list of packages to install, the root file system is created.

Package installation is under control of the package manager (e.g. `dnf/rpm`, `opkg`, or `apt/dpkg`) regardless of whether or not package management is enabled for the target. At the end of the process, if package management is not enabled for the target, the package manager's data files are deleted from the root filesystem. As part of the final stage of package installation, post installation scripts that are part of the packages are run. Any scripts that fail to run on the build host are run on the target when the target system is first booted. If you are using a [read-only root filesystem](#), all the post installation scripts must succeed on the build host during the package installation phase since the root filesystem on the target is read-only.

The final stages of the `do_rootfs` task handle post processing. Post processing includes creation of a manifest file and optimizations.

The manifest file (`.manifest`) resides in the same directory as the root filesystem image. This file lists out, line-by-line, the installed packages. The manifest file is useful for the `testimage` class, for example, to determine whether or not to run specific tests. See the `IMAGE_MANIFEST` variable for additional information.

Optimizing processes that are run across the image include `mklibs`, `prelink`, and any other post-processing commands as defined by the `ROOTFS_POSTPROCESS_COMMAND` variable. The `mklibs` process optimizes the size of the libraries, while the `prelink` process optimizes the dynamic linking of shared libraries to reduce start up time of executables.

After the root filesystem is built, processing begins on the image through the `do_image` task. The build system runs any pre-processing commands as defined by the `IMAGE_PREPROCESS_COMMAND` variable. This variable specifies a list of functions to call before the build system creates the final image output files.

The build system dynamically creates `do_image_*` tasks as needed, based on the image types specified in the `IMAGE_FSTYPES` variable. The process turns everything into an image file or a set of image files and can compress the root filesystem image to reduce the overall size of the image. The formats used for the root filesystem depend on the `IMAGE_FSTYPES` variable. Compression depends on whether the formats support compression.

As an example, a dynamically created task when creating a particular image *type* would take the following form:

```
do_image_type
```

So, if the *type* as specified by the `IMAGE_FSTYPES` were `ext4`, the dynamically generated task would be as follows:

```
do_image_ext4
```

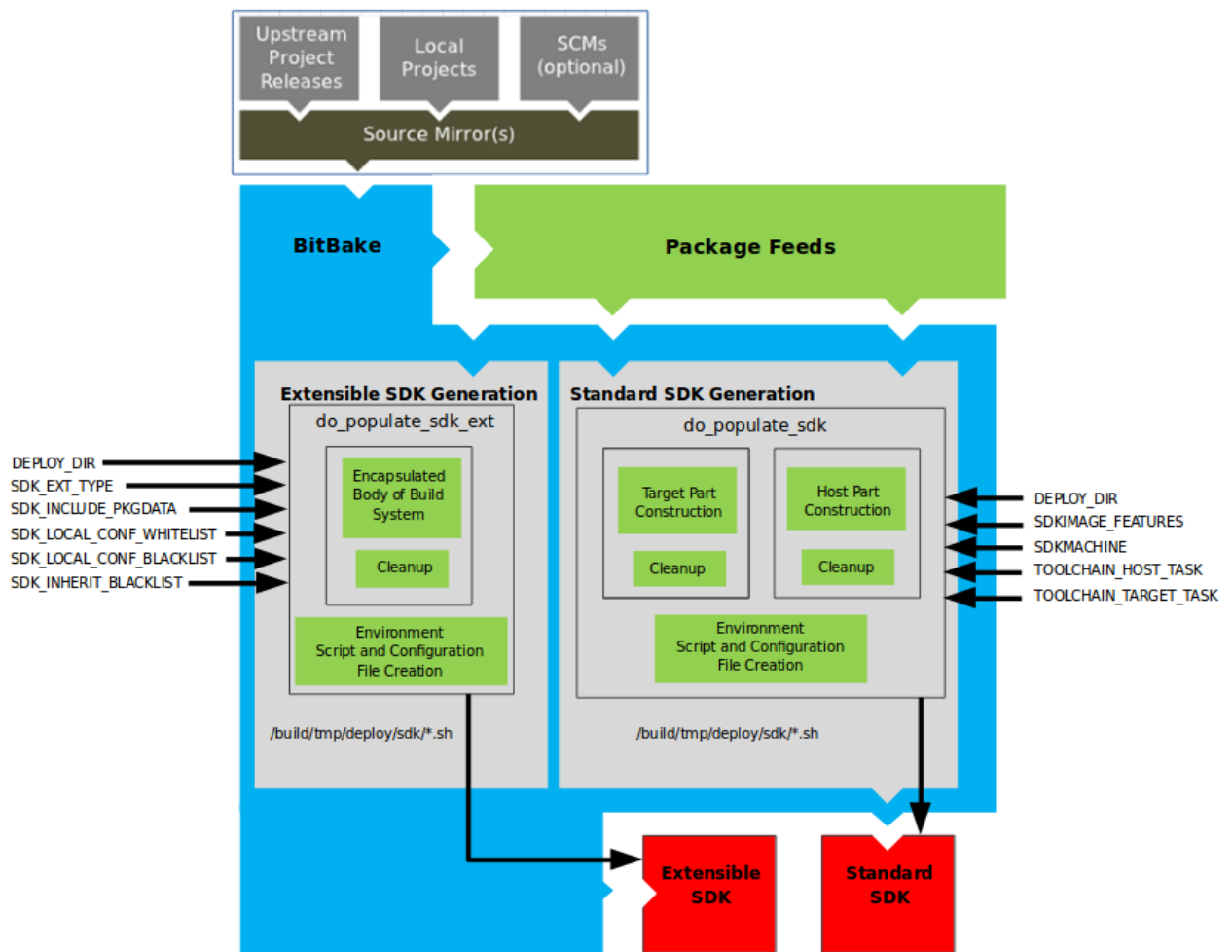
The final task involved in image creation is the `do_image_complete` task. This task completes the image by applying any image post processing as defined through the `IMAGE_POSTPROCESS_COMMAND` variable. The variable specifies a list of functions to call once the build system has created the final image output files.

Note

The entire image generation process is run under `Pseudo`. Running under `Pseudo` ensures that the files in the root filesystem have correct ownership.

4.3.5.6. SDK Generation¶

The OpenEmbedded build system uses BitBake to generate the Software Development Kit (SDK) installer scripts for both the standard SDK and the extensible SDK (eSDK):



Note

For more information on the cross-development toolchain generation, see the "[Cross-Development Toolchain Generation](#)" section. For information on advantages gained when building a cross-development toolchain using the `do_populate_sdk` task, see the "[Building an SDK Installer](#)" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

Like image generation, the SDK script process consists of several stages and depends on many variables. The `do_populate_sdk` and `do_populate_sdk_ext` tasks use these key variables to help create the list of packages to actually install. For information on the variables listed in the figure, see the "[Application Development SDK](#)" section.

The `do_populate_sdk` task helps create the standard SDK and handles two parts: a target part and a host part. The target part is the part built for the target hardware and includes libraries and headers. The host part is the part of the SDK that runs on the `SDKMACHINE`.

The `do_populate_sdk_ext` task helps create the extensible SDK and handles host and target parts differently than its counterpart does for the standard SDK. For the extensible SDK, the task encapsulates the build system, which includes everything needed (host and target) for the SDK.

Regardless of the type of SDK being constructed, the tasks perform some cleanup after which a cross-development environment setup script and any needed configuration files are created. The final output is the Cross-development toolchain installation script (`.sh` file), which includes the environment setup script.

4.3.5.7. Stamp Files and the Rerunning of Tasks¶

For each task that completes successfully, BitBake writes a stamp file into the `STAMPS_DIR` directory. The beginning of the stamp file's filename is determined by the `STAMP` variable, and the end of the name consists of the task's name and current `input_checksum`.

Note

This naming scheme assumes that `BB_SIGNATURE_HANDLER` is "OEBasicHash", which is almost always the case in current OpenEmbedded.

To determine if a task needs to be rerun, BitBake checks if a stamp file with a matching input checksum exists for the task. If such a stamp file exists, the task's output is assumed to exist and still be valid. If the file does not exist, the task is rerun.

Note

The stamp mechanism is more general than the shared state (sstate) cache mechanism described in the "[Setscene Tasks and Shared State](#)" section. BitBake avoids rerunning any task that has a valid stamp file, not just tasks that can be accelerated through the sstate cache.

However, you should realize that stamp files only serve as a marker that some work has been done and that these files do not record task output. The actual task output would usually be somewhere in `TMPDIR` (e.g. in some recipe's `WORKDIR`.) What the sstate cache mechanism adds is a way to cache task output that can then be shared between build machines.

Since `STAMPS_DIR` is usually a subdirectory of `TMPDIR`, removing `TMPDIR` will also remove `STAMPS_DIR`, which means tasks will properly be rerun to repopulate `TMPDIR`.

If you want some task to always be considered "out of date", you can mark it with the `nostamp` varflag. If some other task depends on such a task, then that task will also always be considered out of date, which might not be what you want.

For details on how to view information about a task's signature, see the "[Viewing Task Variable Dependencies](#)" section in the Yocto Project Development Tasks Manual.

4.3.5.8. Setscene Tasks and Shared State¶

The description of tasks so far assumes that BitBake needs to build everything and no available prebuilt objects exist. BitBake does support skipping tasks if prebuilt objects are available. These objects are usually made available in the form of a shared state (sstate) cache.

Note

For information on variables affecting sstate, see the `SSTATE_DIR` and `SSTATE_MIRRORS` variables.

The idea of a setscene task (i.e. `do_taskname_setscene`) is a version of the task where instead of building something, BitBake can skip to the end result and simply place a set of files into specific locations as needed. In some cases, it makes sense to have a setscene task variant (e.g. generating package files in the `do_package_write_*` task). In other cases, it does not make sense (e.g. a `do_patch` task or a `do_unpack` task) since the work involved would be equal to or greater than the underlying task.

In the build system, the common tasks that have setscene variants are `do_package`, `do_package_write_*`, `do_deploy`, `do_packagedata`, and `do_populate_sysroot`. Notice that these tasks represent most of the tasks whose output is an end result.

The build system has knowledge of the relationship between these tasks and other preceding tasks. For example, if BitBake runs `do_populate_sysroot_setscene` for something, it does not make sense to run any of the `do_fetch`, `do_unpack`, `do_patch`, `do_configure`, `do_compile`, and `do_install` tasks. However, if `do_package` needs to be run, BitBake needs to run those other tasks.

It becomes more complicated if everything can come from an sstate cache because some objects are simply not required at all. For example, you do not need a compiler or native tools, such as quilt, if nothing exists to compile or patch. If the `do_package_write_*` packages are available from sstate, BitBake does not need the `do_package` task data.

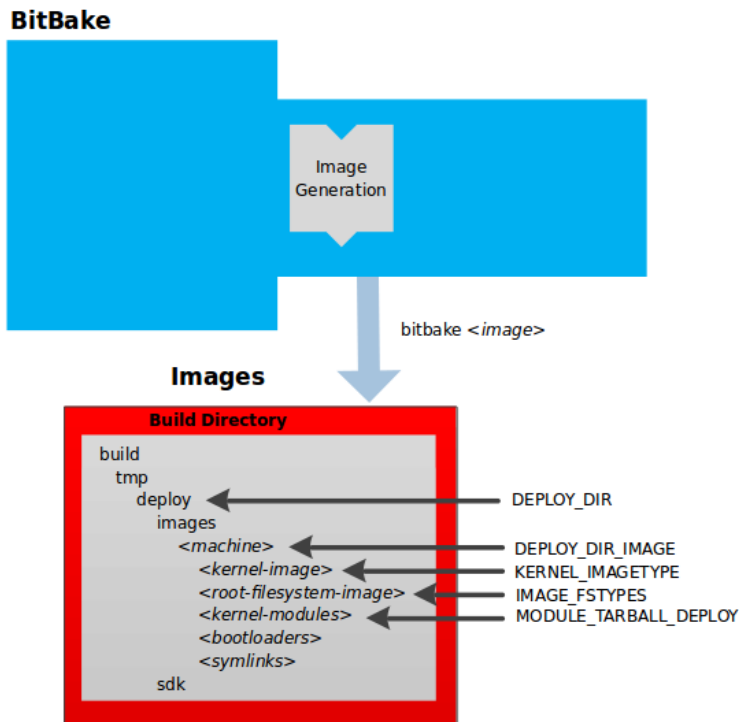
To handle all these complexities, BitBake runs in two phases. The first is the "setscene" stage. During this stage, BitBake first checks the sstate cache for any targets it is planning to build. BitBake does a fast check to see if the object exists rather than a complete download. If nothing exists, the second phase, which is the setscene stage, completes and the main build proceeds.

If objects are found in the sstate cache, the build system works backwards from the end targets specified by the user. For example, if an image is being built, the build system first looks for the packages needed for that image and the tools needed to construct an image. If those are available, the compiler is not needed. Thus, the compiler is not even downloaded. If something was found to be unavailable, or the download or setscene task fails, the build system then tries to install dependencies, such as the compiler, from the cache.

The availability of objects in the sstate cache is handled by the function specified by the `BB_HASHCHECK_FUNCTION` variable and returns a list of available objects. The function specified by the `BB_SETSCENE_DEPVALID` variable is the function that determines whether a given dependency needs to be followed, and whether for any given relationship the function needs to be passed. The function returns a True or False value.

4.3.6. Images¶

The images produced by the build system are compressed forms of the root filesystem and are ready to boot on a target device. You can see from the [general workflow figure](#) that BitBake output, in part, consists of images. This section takes a closer look at this output:



Note

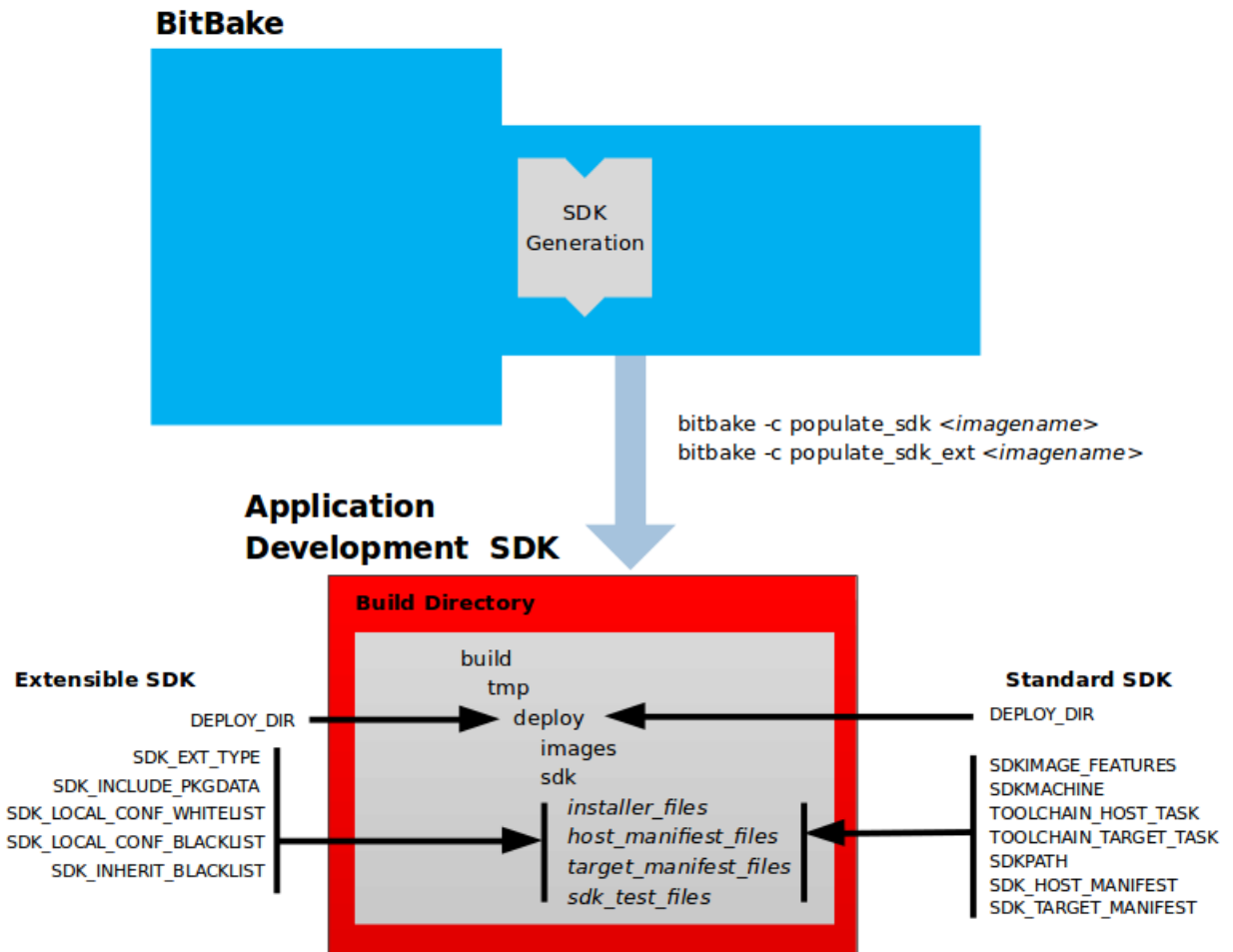
For a list of example images that the Yocto Project provides, see the "[Images](#)" chapter in the Yocto Project Reference Manual.

The build process writes images out to the [Build Directory](#) inside the `tmp/deploy/images/machine/` folder as shown in the figure. This folder contains any files expected to be loaded on the target device. The [DEPLOY_DIR](#) variable points to the `deploy` directory, while the [DEPLOY_DIR_IMAGE](#) variable points to the appropriate directory containing images for the current configuration.

- *kernel-image*: A kernel binary file. The [KERNEL_IMAGETYPE](#) variable determines the naming scheme for the kernel image file. Depending on this variable, the file could begin with a variety of naming strings. The `deploy/images/machine` directory can contain multiple image files for the machine.
- *root-filesystem-image*: Root filesystems for the target device (e.g. *.ext3 or *.bz2 files). The [IMAGE_FSTYPES](#) variable determines the root filesystem image type. The `deploy/images/machine` directory can contain multiple root filesystems for the machine.
- *kernel-modules*: Tarballs that contain all the modules built for the kernel. Kernel module tarballs exist for legacy purposes and can be suppressed by setting the [MODULE_TARBALL_DEPLOY](#) variable to "0". The `deploy/images/machine` directory can contain multiple kernel module tarballs for the machine.
- *bootloaders*: If applicable to the target machine, bootloaders supporting the image. The `deploy/images/machine` directory can contain multiple bootloaders for the machine.
- *symlinks*: The `deploy/images/machine` folder contains a symbolic link that points to the most recently built file for each machine. These links might be useful for external scripts that need to obtain the latest version of each file.

4.3.7. Application Development SDK

In the [general workflow figure](#), the output labeled "Application Development SDK" represents an SDK. The SDK generation process differs depending on whether you build an extensible SDK (e.g. `bitbake -c populate_sdk_ext imagename`) or a standard SDK (e.g. `bitbake -c populate_sdk imagename`). This section takes a closer look at this output:



The specific form of this output is a set of files that includes a self-extracting SDK installer (`*.sh`), host and target manifest files, and files used for SDK testing. When the SDK installer file is run, it installs the SDK. The SDK consists of a cross-development toolchain, a set of libraries and headers, and an SDK environment setup script. Running this installer essentially sets up your cross-development environment. You can think of the cross-toolchain as the "host" part because it runs on the SDK machine. You can think of the libraries and headers as the "target" part because they are built for the target hardware. The environment setup script is added so that you can initialize the environment before using the tools.

Notes

- The Yocto Project supports several methods by which you can set up this cross-development environment. These methods include downloading pre-built SDK installers or building and installing your own SDK installer.
- For background information on cross-development toolchains in the Yocto Project development environment, see the "[Cross-Development Toolchain Generation](#)" section.
- For information on setting up a cross-development environment, see the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

All the output files for an SDK are written to the `deploy/sdk` folder inside the [Build Directory](#) as shown in the previous figure. Depending on the type of SDK, several variables exist that help configure these files. The following list shows the variables associated with an extensible SDK:

- [`DEPLOY\_DIR`](#): Points to the `deploy` directory.
- [`SDK\_EXT\_TYPE`](#): Controls whether or not shared state artifacts are copied into the extensible SDK. By default, all required shared state artifacts are copied into the SDK.
- [`SDK\_INCLUDE\_PKGDATA`](#): Specifies whether or not packagedata is included in the extensible SDK for all recipes in the "world" target.
- [`SDK\_INCLUDE\_TOOLCHAIN`](#): Specifies whether or not the toolchain is included when building the extensible SDK.
- [`SDK\_LOCAL\_CONF\_WHITELIST`](#): A list of variables allowed through from the build system configuration into the extensible SDK configuration.
- [`SDK\_LOCAL\_CONF\_BLACKLIST`](#): A list of variables not allowed through from the build system configuration into the extensible SDK configuration.
- [`SDK\_INHERIT\_BLACKLIST`](#): A list of classes to remove from the [`INHERIT`](#) value globally within the extensible SDK configuration.

This next list, shows the variables associated with a standard SDK:

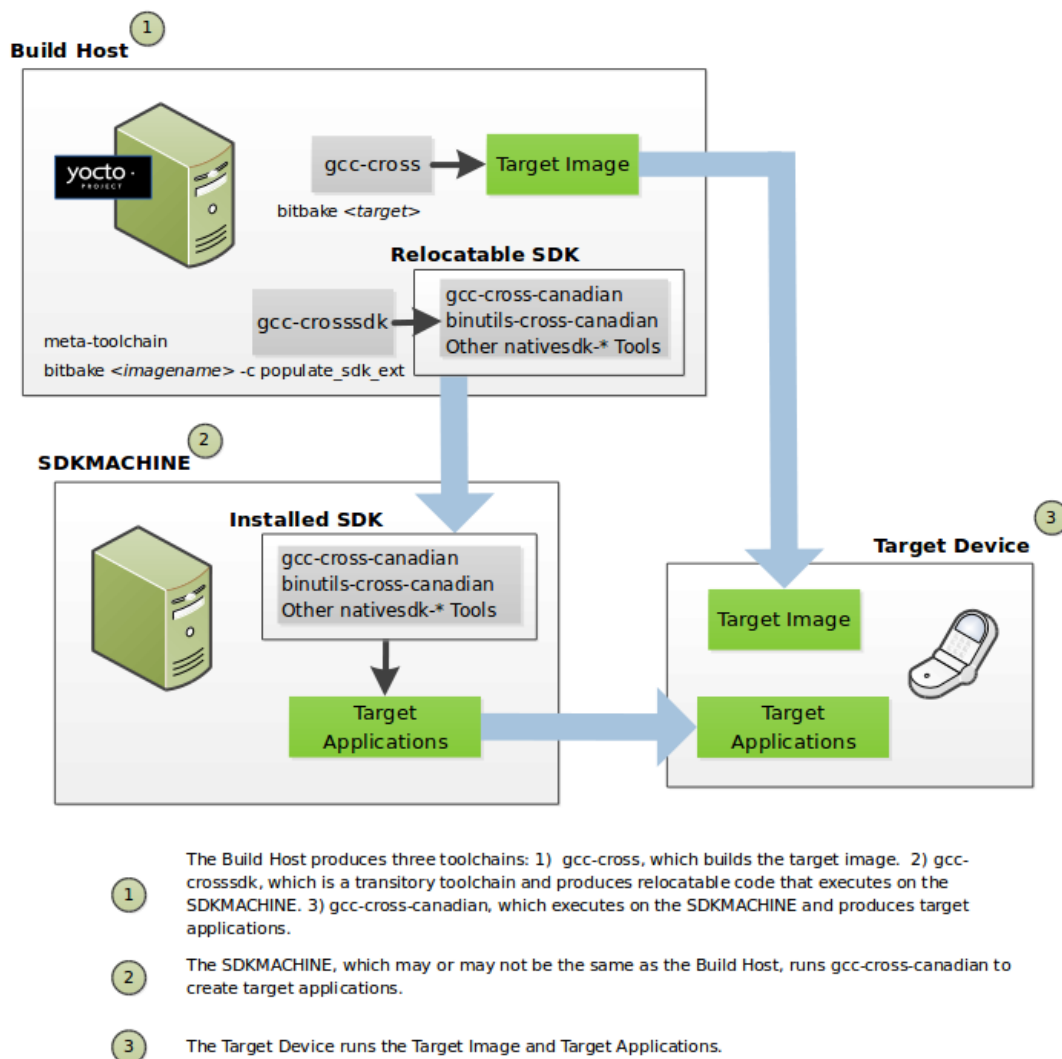
- `DEPLOY_DIR`: Points to the `deploy` directory.
- `SDKMACHINE`: Specifies the architecture of the machine on which the cross-development tools are run to create packages for the target hardware.
- `SDKIMAGE_FEATURES`: Lists the features to include in the "target" part of the SDK.
- `TOOLCHAIN_HOST_TASK`: Lists packages that make up the host part of the SDK (i.e. the part that runs on the `SDKMACHINE`). When you use `bitbake -c populate_sdk imagename` to create the SDK, a set of default packages apply. This variable allows you to add more packages.
- `TOOLCHAIN_TARGET_TASK`: Lists packages that make up the target part of the SDK (i.e. the part built for the target hardware).
- `SDKPATH`: Defines the default SDK installation path offered by the installation script.
- `SDK_HOST_MANIFEST`: Lists all the installed packages that make up the host part of the SDK. This variable also plays a minor role for extensible SDK development as well. However, it is mainly used for the standard SDK.
- `SDK_TARGET_MANIFEST`: Lists all the installed packages that make up the target part of the SDK. This variable also plays a minor role for extensible SDK development as well. However, it is mainly used for the standard SDK.

4.4. Cross-Development Toolchain Generation¶

The Yocto Project does most of the work for you when it comes to creating [cross-development toolchains](#). This section provides some technical background on how cross-development toolchains are created and used. For more information on toolchains, you can also see the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

In the Yocto Project development environment, cross-development toolchains are used to build images and applications that run on the target hardware. With just a few commands, the OpenEmbedded build system creates these necessary toolchains for you.

The following figure shows a high-level build environment regarding toolchain construction and use.



Most of the work occurs on the Build Host. This is the machine used to build images and generally work within the the Yocto Project environment. When you run `BitBake` to create an image, the OpenEmbedded build system uses the host `gcc` compiler to bootstrap a cross-compiler named `gcc-cross`. The `gcc-cross` compiler is what BitBake uses to compile source files when creating the target image. You can think of `gcc-cross` simply as an automatically generated cross-compiler that is used internally within BitBake only.

Note

The extensible SDK does not use `gcc-cross-canadian` since this SDK ships a copy of the OpenEmbedded build system and the sysroot within it contains `gcc-cross`.

The chain of events that occurs when `gcc-cross` is bootstrapped is as follows:

```
gcc -> binutils-cross -> gcc-cross-initial -> linux-libc-headers -> glibc-initial -> glibc -> gcc-cross -> gcc-runtime
```

- `gcc`: The build host's GNU Compiler Collection (GCC).
- `binutils-cross`: The bare minimum binary utilities needed in order to run the `gcc-cross-initial` phase of the bootstrap operation.
- `gcc-cross-initial`: An early stage of the bootstrap process for creating the cross-compiler. This stage builds enough of the `gcc-cross`, the C library, and other pieces needed to finish building the final cross-compiler in later stages. This tool is a "native" package (i.e. it is designed to run on the build host).
- `linux-libc-headers`: Headers needed for the cross-compiler.
- `glibc-initial`: An initial version of the Embedded GNU C Library (GLIBC) needed to bootstrap `glibc`.
- `glibc`: The GNU C Library.
- `gcc-cross`: The final stage of the bootstrap process for the cross-compiler. This stage results in the actual cross-compiler that BitBake uses when it builds an image for a targeted device.

Note

If you are replacing this cross compiler toolchain with a custom version, you must replace `gcc-cross`.

This tool is also a "native" package (i.e. it is designed to run on the build host).

- `gcc-runtime`: Runtime libraries resulting from the toolchain bootstrapping process. This tool produces a binary that consists of the runtime libraries need for the targeted device.

You can use the OpenEmbedded build system to build an installer for the relocatable SDK used to develop applications. When you run the installer, it installs the toolchain, which contains the development tools (e.g., `gcc-cross-canadian`, `binutils-cross-canadian`, and other `nativesdk-*` tools), which are tools native to the SDK (i.e. native to [SDK_ARCH](#)), you need to cross-compile and test your software. The figure shows the commands you use to easily build out this toolchain. This cross-development toolchain is built to execute on the [SDKMACHINE](#), which might or might not be the same machine as the Build Host.

Note

If your target architecture is supported by the Yocto Project, you can take advantage of pre-built images that ship with the Yocto Project and already contain cross-development toolchain installers.

Here is the bootstrap process for the relocatable toolchain:

```
gcc -> binutils-crosssdk -> gcc-crosssdk-initial -> linux-libc-headers ->
glibc-initial -> nativesdk-glibc -> gcc-crosssdk -> gcc-cross-canadian
```

- `gcc`: The build host's GNU Compiler Collection (GCC).
- `binutils-crosssdk`: The bare minimum binary utilities needed in order to run the `gcc-crosssdk-initial` phase of the bootstrap operation.
- `gcc-crosssdk-initial`: An early stage of the bootstrap process for creating the cross-compiler. This stage builds enough of the `gcc-crosssdk` and supporting pieces so that the final stage of the bootstrap process can produce the finished cross-compiler. This tool is a "native" binary that runs on the build host.
- `linux-libc-headers`: Headers needed for the cross-compiler.
- `glibc-initial`: An initial version of the Embedded GLIBC needed to bootstrap `nativesdk-glibc`.
- `nativesdk-glibc`: The Embedded GLIBC needed to bootstrap the `gcc-crosssdk`.
- `gcc-crosssdk`: The final stage of the bootstrap process for the relocatable cross-compiler. The `gcc-crosssdk` is a transitory compiler and never leaves the build host. Its purpose is to help in the bootstrap process to create the eventual `gcc-cross-canadian` compiler, which is relocatable. This tool is also a "native" package (i.e. it is designed to run on the build host).
- `gcc-cross-canadian`: The final relocatable cross-compiler. When run on the [SDKMACHINE](#), this tool produces executable code that runs on the target device. Only one cross-canadian compiler is produced per architecture since they can be targeted at different processor optimizations using configurations passed to the compiler through the compile commands. This circumvents the need for multiple compilers and thus reduces the size of the toolchains.

Note

For information on advantages gained when building a cross-development toolchain installer, see the "[Building an SDK Installer](#)" appendix in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

4.5. Shared State Cache¶

By design, the OpenEmbedded build system builds everything from scratch unless [BitBake](#) can determine that parts do not need to be rebuilt. Fundamentally, building from scratch is attractive as it means all parts are built fresh and no possibility of stale data exists that can cause problems. When developers hit problems, they typically default back to building from scratch so they have a known state from the start.

Building an image from scratch is both an advantage and a disadvantage to the process. As mentioned in the previous paragraph, building from scratch ensures that everything is current and starts from a known state. However, building from scratch also takes much longer as it generally means rebuilding things that do not necessarily need to be rebuilt.

The Yocto Project implements shared state code that supports incremental builds. The implementation of the shared state code answers the following questions that were fundamental roadblocks within the OpenEmbedded incremental build support system:

- What pieces of the system have changed and what pieces have not changed?
- How are changed pieces of software removed and replaced?
- How are pre-built components that do not need to be rebuilt from scratch used when they are available?

For the first question, the build system detects changes in the "inputs" to a given task by creating a checksum (or signature) of the task's inputs. If the checksum changes, the system assumes the inputs have changed and the task needs to be rerun. For the second question, the shared state (sstate) code tracks which tasks add which output to the build process. This means the output from a given task can be removed, upgraded or otherwise manipulated. The third question is partly addressed by the solution for the second question assuming the build system can fetch the sstate objects from remote locations and install them if they are deemed to be valid.

Notes

- The build system does not maintain [PR](#) information as part of the shared state packages. Consequently, considerations exist that affect maintaining shared state feeds. For information on how the build system works with packages and can track incrementing [PR](#) information, see the "[Automatically Incrementing a Binary Package Revision Number](#)" section in the Yocto Project Development Tasks Manual.
- The code in the build system that supports incremental builds is not simple code. For techniques that help you work around issues related to shared state code, see the "[Viewing Metadata Used to Create the Input Signature of a Shared State Task](#)" and "[Invalidating Shared State to Force a Task to Run](#)" sections both in the Yocto Project Development Tasks Manual.

The rest of this section goes into detail about the overall incremental build architecture, the checksums (signatures), and shared state.

4.5.1. Overall Architecture¶

When determining what parts of the system need to be built, BitBake works on a per-task basis rather than a per-recipe basis. You might wonder why using a per-task basis is preferred over a per-recipe basis. To help explain, consider having the IPK packaging backend enabled and then switching to DEB. In this case, the [do_install](#) and [do_package](#) task outputs are still valid. However, with a per-recipe approach, the build would not include the `.deb` files. Consequently, you would have to invalidate the whole build and rerun it. Rerunning everything is not the best solution. Also, in this case, the core must be "taught" much about specific tasks. This methodology does not scale well and does not allow users to easily add new tasks in layers or as external recipes without touching the packaged-staging core.

4.5.2. Checksums (Signatures)¶

The shared state code uses a checksum, which is a unique signature of a task's inputs, to determine if a task needs to be run again. Because it is a change in a task's inputs that triggers a rerun, the process needs to detect all the inputs to a given task. For shell tasks, this turns out to be fairly easy because the build process generates a "run" shell script for each task and it is possible to create a checksum that gives you a good idea of when the task's data changes.

To complicate the problem, there are things that should not be included in the checksum. First, there is the actual specific build path of a given task - the [WORKDIR](#). It does not matter if the work directory changes because it should not affect the output for target packages. Also, the build process has the objective of making native or cross packages relocatable.

Note

Both native and cross packages run on the [build host](#). However, cross packages generate output for the target architecture.

The checksum therefore needs to exclude [WORKDIR](#). The simplistic approach for excluding the work directory is to set [WORKDIR](#) to some fixed value and create the checksum for the "run" script.

Another problem results from the "run" scripts containing functions that might or might not get called. The incremental build solution contains code that figures out dependencies between shell functions. This code is used to prune the "run" scripts down to the minimum set, thereby alleviating this problem and making the "run" scripts much more readable as a bonus.

So far, solutions for shell scripts exist. What about Python tasks? The same approach applies even though these tasks are more difficult. The process needs to figure out what variables a Python function accesses and what functions it calls. Again, the incremental build solution contains code that first figures out the variable and function dependencies, and then creates a checksum for the data used as the input to the task.

Like the `WORKDIR` case, situations exist where dependencies should be ignored. For these situations, you can instruct the build process to ignore a dependency by using a line like the following:

```
PACKAGE_ARCHS[vardepsexclude] = "MACHINE"
```

This example ensures that the `PACKAGE_ARCHS` variable does not depend on the value of `MACHINE`, even if it does reference it.

Equally, there are cases where you need to add dependencies BitBake is not able to find. You can accomplish this by using a line like the following:

```
PACKAGE_ARCHS[vardeps] = "MACHINE"
```

This example explicitly adds the `MACHINE` variable as a dependency for `PACKAGE_ARCHS`.

As an example, consider a case with in-line Python where BitBake is not able to figure out dependencies. When running in debug mode (i.e. using `-DDD`), BitBake produces output when it discovers something for which it cannot figure out dependencies. The Yocto Project team has currently not managed to cover those dependencies in detail and is aware of the need to fix this situation.

Thus far, this section has limited discussion to the direct inputs into a task. Information based on direct inputs is referred to as the "basehash" in the code. However, the question of a task's indirect inputs still exists - items already built and present in the [Build Directory](#). The checksum (or signature) for a particular task needs to add the hashes of all the tasks on which the particular task depends. Choosing which dependencies to add is a policy decision. However, the effect is to generate a master checksum that combines the basehash and the hashes of the task's dependencies.

At the code level, a variety of ways exist by which both the basehash and the dependent task hashes can be influenced. Within the BitBake configuration file, you can give BitBake some extra information to help it construct the basehash. The following statement effectively results in a list of global variable dependency excludes (i.e. variables never included in any checksum):

```
BB_HASHBASE_WHITELIST ?= "TMPDIR FILE PATH PWD BB_TASKHASH BBPATH DL_DIR \
SSSTATE_DIR THISDIR FILESEXTRAPATHS FILE_DIRNAME HOME LOGNAME SHELL TERM \
USER FILESPATH STAGING_DIR_HOST STAGING_DIR_TARGET COREBASE PRSERV_HOST \
PRSERV_DUMPDIR PRSERV_DUMPFILE PRSERV_LOCKDOWN PARALLEL_MAKE \
CCACHE_DIR EXTERNAL_TOOLCHAIN CCACHE CCACHE_DISABLE LICENSE_PATH SDKPKGSUFFIX"
```

The previous example excludes `WORKDIR` since that variable is actually constructed as a path within `TMPDIR`, which is on the whitelist.

The rules for deciding which hashes of dependent tasks to include through dependency chains are more complex and are generally accomplished with a Python function. The code in `meta/lib/oe/sstatesig.py` shows two examples of this and also illustrates how you can insert your own policy into the system if so desired. This file defines the two basic signature generators [OE-Core](#) uses: "OEBasic" and "OEBasicHash". By default, a dummy "noop" signature handler is enabled in BitBake. This means that behavior is unchanged from previous versions. OE-Core uses the "OEBasicHash" signature handler by default through this setting in the `bitbake.conf` file:

```
BB_SIGNATURE_HANDLER ?= "OEBasicHash"
```

The "OEBasicHash" `BB_SIGNATURE_HANDLER` is the same as the "OEBasic" version but adds the task hash to the [stamp files](#). This results in any metadata change that changes the task hash, automatically causing the task to be run again. This removes the need to bump `PR` values, and changes to metadata automatically ripple across the build.

It is also worth noting that the end result of these signature generators is to make some dependency and hash information available to the build. This information includes:

- `BB_BASEHASH_task-taskname`: The base hashes for each task in the recipe.
- `BB_BASEHASH_filename:taskname`: The base hashes for each dependent task.
- `BBHASHDEPS_filename:taskname`: The task dependencies for each task.
- `BB_TASKHASH`: The hash of the currently running task.

4.5.3. Shared State

Checksums and dependencies, as discussed in the previous section, solve half the problem of supporting a shared state. The other half of the problem is being able to use checksum information during the build and being able to reuse or rebuild specific components.

The `sstate` class is a relatively generic implementation of how to "capture" a snapshot of a given task. The idea is that the build process does not care about the source of a task's output. Output could be freshly built or it could be downloaded and unpacked from somewhere. In other words, the build process does not need to worry about its origin.

Two types of output exist. One type is just about creating a directory in `WORKDIR`. A good example is the output of either `do_install` or `do_package`. The other type of output occurs when a set of data is merged into a shared directory tree such as the `sysroot`.

The Yocto Project team has tried to keep the details of the implementation hidden in `sstate` class. From a user's perspective, adding shared state wrapping to a task is as simple as this `do_deploy` example taken from the `deploy` class:

```
DEPLOYDIR = "${WORKDIR}/deploy-${PN}"
SSSTATETASKS += "do_deploy"
do_deploy[sstate-inputdirs] = "${DEPLOYDIR}"
do_deploy[sstate-outputdirs] = "${DEPLOY_DIR_IMAGE}"

python do_deploy_setscene () {
    sstate_setscene(d)
}

addtask do_deploy_setscene
do_deploy[dirs] = "${DEPLOYDIR} ${B}"
do_deploy[stamp-extra-info] = "${MACHINE_ARCH}"
```

The following list explains the previous example:

- Adding "do_deploy" to SSTATETASKS adds some required sstate-related processing, which is implemented in the `sstate` class, to before and after the `do_deploy` task.
- The `do_deploy[sstate-inputdirs] = "${DEPLOYDIR}"` declares that `do_deploy` places its output in `${DEPLOYDIR}` when run normally (i.e. when not using the sstate cache). This output becomes the input to the shared state cache.
- The `do_deploy[sstate-outputdirs] = "${DEPLOY_DIR_IMAGE}"` line causes the contents of the shared state cache to be copied to `${DEPLOY_DIR_IMAGE}`.

Note

If `do_deploy` is not already in the shared state cache or if its input checksum (signature) has changed from when the output was cached, the task runs to populate the shared state cache, after which the contents of the shared state cache is copied to `${DEPLOY_DIR_IMAGE}`. If `do_deploy` is in the shared state cache and its signature indicates that the cached output is still valid (i.e. if no relevant task inputs have changed), then the contents of the shared state cache copies directly to `${DEPLOY_DIR_IMAGE}` by the `do_deploy_setscene` task instead, skipping the `do_deploy` task.

- The following task definition is glue logic needed to make the previous settings effective:

```
python do_deploy_setscene () {
    sstate_setscene(d)
}
addtask do_deploy_setscene
```

`sstate_setscene()` takes the flags above as input and accelerates the `do_deploy` task through the shared state cache if possible. If the task was accelerated, `sstate_setscene()` returns True. Otherwise, it returns False, and the normal `do_deploy` task runs. For more information, see the "[setscene](#)" section in the BitBake User Manual.

- The `do_deploy[dirs] = "${DEPLOYDIR} ${B}"` line creates `${DEPLOYDIR}` and `${B}` before the `do_deploy` task runs, and also sets the current working directory of `do_deploy` to `${B}`. For more information, see the "[Variable Flags](#)" section in the BitBake User Manual.

Note

In cases where `sstate-inputdirs` and `sstate-outputdirs` would be the same, you can use `sstate-plaindirs`. For example, to preserve the `${PKGDEST}` and `${PKGDEST}` output from the `do_package` task, use the following:

```
do_package[sstate-plaindirs] = "${PKGDEST} ${PKGDEST}"
```

- The `do_deploy[stamp-extra-info] = "${MACHINE_ARCH}"` line appends extra metadata to the [stamp file](#). In this case, the metadata makes the task specific to a machine's architecture. See "[The Task List](#)" section in the BitBake User Manual for more information on the `stamp-extra-info` flag.
- `sstate-inputdirs` and `sstate-outputdirs` can also be used with multiple directories. For example, the following declares `PKGDESTWORK` and `SHLIBWORK` as shared state input directories, which populates the shared state cache, and `PKGDATA_DIR` and `SHLIBSDIR` as the corresponding shared state output directories:

```
do_package[sstate-inputdirs] = "${PKGDESTWORK} ${SHLIBWORKDIR}"
do_package[sstate-outputdirs] = "${PKGDATA_DIR} ${SHLIBSDIR}"
```

- These methods also include the ability to take a lockfile when manipulating shared state directory structures, for cases where file additions or removals are sensitive:

```
do_package[sstate-lockfile] = "${PACKAGELOCK}"
```

Behind the scenes, the shared state code works by looking in [SSTATE_DIR](#) and [SSTATE_MIRRORS](#) for shared state files. Here is an example:

```
SSTATE_MIRRORS ?= "\
file://.* http://someserver.tld/share/sstate/PATH;downloadfilename=PATH \n \
file://.* file:///some/local/dir/sstate/PATH"
```

Note

The shared state directory (`SSTATE_DIR`) is organized into two-character subdirectories, where the subdirectory names are based on the first two characters of the hash. If the shared state directory structure for a mirror has the same structure as `SSTATE_DIR`, you must specify "PATH" as part of the URI to enable the build system to map to the appropriate subdirectory.

The shared state package validity can be detected just by looking at the filename since the filename contains the task checksum (or signature) as described earlier in this section. If a valid shared state package is found, the build process downloads it and uses it to accelerate the task.

The build processes use the `*_setscene` tasks for the task acceleration phase. BitBake goes through this phase before the main execution code and tries to accelerate any tasks for which it can find shared state packages. If a shared state package for a task is available, the shared state package is used. This means the task and any tasks on which it is dependent are not executed.

As a real world example, the aim is when building an IPK-based image, only the `do_package_write_ipk` tasks would have their shared state packages fetched and extracted. Since the sysroot is not used, it would never get extracted. This is another reason why a task-based approach is preferred over a recipe-based approach, which would have to install the output from every task.

4.6. Automatically Added Runtime Dependencies¶

The OpenEmbedded build system automatically adds common types of runtime dependencies between packages, which means that you do not need to explicitly declare the packages using `RDEPENDS`. Three automatic mechanisms exist (`shlibdeps`, `pcdeps`, and `depchains`) that handle shared libraries, package configuration (pkg-config) modules, and `-dev` and `-dbg` packages, respectively. For other types of runtime dependencies, you must manually declare the dependencies.

- `shlibdeps`: During the `do_package` task of each recipe, all shared libraries installed by the recipe are located. For each shared library, the package that contains the shared library is registered as providing the shared library. More specifically, the package is registered as providing the `soname` of the library. The resulting shared-library-to-package mapping is saved globally in `PKGDATA_DIR` by the `do_packagedata` task.

Simultaneously, all executables and shared libraries installed by the recipe are inspected to see what shared libraries they link against. For each shared library dependency that is found, `PKGDATA_DIR` is queried to see if some package (likely from a different recipe) contains the shared library. If such a package is found, a runtime dependency is added from the package that depends on the shared library to the package that contains the library.

The automatically added runtime dependency also includes a version restriction. This version restriction specifies that at least the current version of the package that provides the shared library must be used, as if `"package (>= version)"` had been added to `RDEPENDS`. This forces an upgrade of the package containing the shared library when installing the package that depends on the library, if needed.

If you want to avoid a package being registered as providing a particular shared library (e.g. because the library is for internal use only), then add the library to `PRIVATE_LIBS` inside the package's recipe.

- `pcdeps`: During the `do_package` task of each recipe, all pkg-config modules (`*.pc` files) installed by the recipe are located. For each module, the package that contains the module is registered as providing the module. The resulting module-to-package mapping is saved globally in `PKGDATA_DIR` by the `do_packagedata` task.

Simultaneously, all pkg-config modules installed by the recipe are inspected to see what other pkg-config modules they depend on. A module is seen as depending on another module if it contains a "Requires:" line that specifies the other module. For each module dependency, `PKGDATA_DIR` is queried to see if some package contains the module. If such a package is found, a runtime dependency is added from the package that depends on the module to the package that contains the module.

Note

The `pcdeps` mechanism most often infers dependencies between `-dev` packages.

- `depchains`: If a package `foo` depends on a package `bar`, then `foo-dev` and `foo-dbg` are also made to depend on `bar-dev` and `bar-dbg`, respectively. Taking the `-dev` packages as an example, the `bar-dev` package might provide headers and shared library symlinks needed by `foo-dev`, which shows the need for a dependency between the packages.

The dependencies added by `depchains` are in the form of `RRECOMMENDS`.

Note

By default, `foo-dev` also has an `RDEPENDS`-style dependency on `foo`, because the default value of `RDEPENDS_${PN}-dev` (set in `bitbake.conf`) includes `"${PN}"`.

To ensure that the dependency chain is never broken, `-dev` and `-dbg` packages are always generated by default, even if the packages turn out to be empty. See the `ALLOW_EMPTY` variable for more information.

The `do_package` task depends on the `do_packagedata` task of each recipe in `DEPENDS` through use of a `[deptask]` declaration, which guarantees that the required shared-library/module-to-package mapping information will be available when needed as long as `DEPENDS` has been correctly set.

4.7. Fakeroot and Pseudo¶

Some tasks are easier to implement when allowed to perform certain operations that are normally reserved for the root user (e.g. `do_install`, `do_package_write*`, `do_rootfs`, and `do_image*`). For example, the `do_install` task benefits from being able to set the UID and GID of installed files to arbitrary values.

One approach to allowing tasks to perform root-only operations would be to require `BitBake` to run as root. However, this method is cumbersome and has security issues. The approach that is actually used is to run tasks that benefit from root privileges in a "fake" root environment. Within this environment, the task and its child processes believe that they are running as the root user, and see an internally consistent view of the filesystem. As long as generating the final output (e.g. a package or an image) does not require root privileges, the fact that some earlier steps ran in a fake root environment does not cause problems.

The capability to run tasks in a fake root environment is known as `"fakeroot"`, which is derived from the BitBake keyword/variable flag that requests a fake root environment for a task.

In the [OpenEmbedded build system](#), the program that implements fakeroot is known as [Pseudo](#). Pseudo overrides system calls by using the environment variable `LD_PRELOAD`, which results in the illusion of running as root. To keep track of "fake" file ownership and permissions resulting from operations that require root permissions, Pseudo uses an SQLite 3 database. This database is stored in `${WORKDIR}/pseudo/files.db` for individual recipes. Storing the database in a file as opposed to in memory gives persistence between tasks and builds, which is not accomplished using fakeroot.

Caution

If you add your own task that manipulates the same files or directories as a fakeroot task, then that task also needs to run under fakeroot. Otherwise, the task cannot run root-only operations, and cannot see the fake file ownership and permissions set by the other task. You need to also add a dependency on `virtual/fakeroot-native:do_populate_sysroot`, giving the following:

```
fakeroot do_mytask () {
    ...
}
do_mytask[depends] += "virtual/fakeroot-native:do_populate_sysroot"
```

For more information, see the [FAKEROOT*](#) variables in the BitBake User Manual. You can also reference the ["Why Not Fakeroot?"](#) article for background information on Fakeroot and Pseudo.



Yocto Project Development Tasks Manual

Chapter 5. The Yocto Project Development Tasks Manual¶

[5.1. Welcome](#)

[5.2. Other Information](#)

5.1. Welcome¶

Welcome to the Yocto Project Development Tasks Manual! This manual provides relevant procedures necessary for developing in the Yocto Project environment (i.e. developing embedded Linux images and user-space applications that run on targeted devices). The manual groups related procedures into higher-level sections. Procedures can consist of high-level steps or low-level steps depending on the topic.

This manual provides the following:

- Procedures that help you get going with the Yocto Project. For example, procedures that show you how to set up a build host and work with the Yocto Project source repositories.
- Procedures that show you how to submit changes to the Yocto Project. Changes can be improvements, new features, or bug fixes.
- Procedures related to "everyday" tasks you perform while developing images and applications using the Yocto Project. For example, procedures to create a layer, customize an image, write a new recipe, and so forth.

This manual does not provide the following:

- Redundant Step-by-step Instructions: For example, the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual contains detailed instructions on how to install an SDK, which is used to develop applications for target hardware.
- Reference or Conceptual Material: This type of material resides in an appropriate reference manual. For example, system variables are documented in the Yocto Project Reference Manual.
- Detailed Public Information Not Specific to the Yocto Project: For example, exhaustive information on how to use the Source Control Manager Git is better covered with Internet searches and official Git Documentation than through the Yocto Project documentation.

5.2. Other Information¶

Because this manual presents information for many different topics, supplemental information is recommended for full comprehension. For introductory information on the Yocto Project, see the [Yocto Project Website](#). If you want to build an image with no knowledge of Yocto Project as a way of quickly testing it out, see the Yocto Project Quick Build document.

For a comprehensive list of links and other documentation, see the ["Links and Related Documentation"](#) section in the Yocto Project Reference Manual.

Chapter 6. Setting Up to Use the Yocto Project¶

- [6.1. Creating a Team Development Environment](#)
- [6.2. Preparing the Build Host](#)
 - [6.2.1. Setting Up a Native Linux Host](#)
 - [6.2.2. Setting Up to Use CROss PlatformS \(CROPS\)](#)
- [6.3. Locating Yocto Project Source Files](#)
 - [6.3.1. Accessing Source Repositories](#)
 - [6.3.2. Accessing Index of Releases](#)
 - [6.3.3. Using the Downloads Page](#)
 - [6.3.4. Accessing Nightly Builds](#)
- [6.4. Cloning and Checking Out Branches](#)
 - [6.4.1. Cloning the poky Repository](#)
 - [6.4.2. Checking Out by Branch in Poky](#)
 - [6.4.3. Checking Out by Tag in Poky](#)

This chapter provides procedures related to getting set up to use the Yocto Project. You can learn about creating a team environment that develops using the Yocto Project, how to set up a [build host](#), how to locate Yocto Project source repositories, and how to create local Git repositories.

6.1. Creating a Team Development Environment¶

It might not be immediately clear how you can use the Yocto Project in a team development environment, or how to scale it for a large team of developers. You can adapt the Yocto Project to many different use cases and scenarios. However, this flexibility could cause difficulties if you are trying to create a working setup that scales across a large team.

To help you understand how to set up this type of environment, this section presents a procedure that gives you information that can help you get the results you want. The procedure is high-level and presents some of the project's most successful experiences, practices, solutions, and available technologies that have proved to work well in the past. Keep in mind, the procedure here is a starting point. You can build off these steps and customize the procedure to fit any particular working environment and set of practices.

1. **Determine Who is Going to be Developing:** You need to understand who is going to be doing anything related to the Yocto Project and determine their roles. Making this determination is essential to completing steps two and three, which are to get your equipment together and set up your development environment's hardware topology.

The following roles exist:

- **Application Developer:** This type of developer does application level work on top of an existing software stack.
 - **Core System Developer:** This type of developer works on the contents of the operating system image itself.
 - **Build Engineer:** This type of developer manages Autobuilders and releases. Not all environments need a Build Engineer.
 - **Test Engineer:** This type of developer creates and manages automated tests that are used to ensure all application and core system development meets desired quality standards.
2. **Gather the Hardware:** Based on the size and make-up of the team, get the hardware together. Ideally, any development, build, or test engineer uses a system that runs a supported Linux distribution. These systems, in general, should be high performance (e.g. dual, six-core Xeons with 24 Gbytes of RAM and plenty of disk space). You can help ensure efficiency by having any machines used for testing or that run Autobuilders be as high performance as possible.
 3. **Understand the Hardware Topology of the Environment:** Once you understand the hardware involved and the make-up of the team, you can understand the hardware topology of the development environment. You can get a visual idea of the machines and their roles across the development environment.
 4. **Use Git as Your Source Control Manager (SCM):** Keeping your [Metadata](#) (i.e. recipes, configuration files, classes, and so forth) and any software you are developing under the control of an SCM system that is compatible with the OpenEmbedded build system is advisable. Of the SCMs BitBake supports, the Yocto Project team strongly recommends using [Git](#). Git is a distributed system that is easy to backup, allows you to work remotely, and then connects back to the infrastructure.

Note

For information about BitBake, see the BitBake User Manual.

It is relatively easy to set up Git services and create infrastructure like <http://git.yoctoproject.org>, which is based on server software called `gitolite` with `cgkit` being used to generate the web interface that lets you view the repositories. The `gitolite` software identifies users using SSH keys and allows branch-based access controls to repositories that you can control as little or as much as necessary.

Note

The setup of these services is beyond the scope of this manual. However, sites such as the following exist that describe how to perform setup:

- [Git documentation](#): Describes how to install `gitolite` on the server.
- [Gitolite](#): Information for `gitolite`.
- [Interfaces, frontends, and tools](#): Documentation on how to create interfaces and frontends for Git.

5. **Set up the Application Development Machines:** As mentioned earlier, application developers are creating applications on top of existing software stacks. Following are some best practices for setting up machines used for application development:

- Use a pre-built toolchain that contains the software stack itself. Then, develop the application code on top of the stack. This method works well for small numbers of relatively isolated applications.

- Keep your cross-development toolchains updated. You can do this through provisioning either as new toolchain downloads or as updates through a package update mechanism using `opkg` to provide updates to an existing toolchain. The exact mechanics of how and when to do this depend on local policy.
 - Use multiple toolchains installed locally into different locations to allow development across versions.
6. **Set up the Core Development Machines:** As mentioned earlier, core developers work on the contents of the operating system itself. Following are some best practices for setting up machines used for developing images:
- Have the [OpenEmbedded build system](#) available on the developer workstations so developers can run their own builds and directly rebuild the software stack.
 - Keep the core system unchanged as much as possible and do your work in layers on top of the core system. Doing so gives you a greater level of portability when upgrading to new versions of the core system or Board Support Packages (BSPs).
 - Share layers amongst the developers of a particular project and contain the policy configuration that defines the project.
7. **Set up an Autobuilder:** Autobuilders are often the core of the development environment. It is here that changes from individual developers are brought together and centrally tested. Based on this automated build and test environment, subsequent decisions about releases can be made. Autobuilders also allow for "continuous integration" style testing of software components and regression identification and tracking.
- See "[Yocto Project Autobuilder](#)" for more information and links to buildbot. The Yocto Project team has found this implementation works well in this role. A public example of this is the Yocto Project Autobuilders, which the Yocto Project team uses to test the overall health of the project.
- The features of this system are:
- Highlights when commits break the build.
 - Populates an [sstate cache](#) from which developers can pull rather than requiring local builds.
 - Allows commit hook triggers, which trigger builds when commits are made.
 - Allows triggering of automated image booting and testing under the QuickEMUlator (QEMU).
 - Supports incremental build testing and from-scratch builds.
 - Shares output that allows developer testing and historical regression investigation.
 - Creates output that can be used for releases.
 - Allows scheduling of builds so that resources can be used efficiently.
8. **Set up Test Machines:** Use a small number of shared, high performance systems for testing purposes. Developers can use these systems for wider, more extensive testing while they continue to develop locally using their primary development system.
9. **Document Policies and Change Flow:** The Yocto Project uses a hierarchical structure and a pull model. Scripts exist to create and send pull requests (i.e. `create-pull-request` and `send-pull-request`). This model is in line with other open source projects where maintainers are responsible for specific areas of the project and a single maintainer handles the final "top-of-tree" merges.

Note

You can also use a more collective push model. The `gitolite` software supports both the push and pull models quite easily.

As with any development environment, it is important to document the policy used as well as any main project guidelines so they are understood by everyone. It is also a good idea to have well structured commit messages, which are usually a part of a project's guidelines. Good commit messages are essential when looking back in time and trying to understand why changes were made.

If you discover that changes are needed to the core layer of the project, it is worth sharing those with the community as soon as possible. Chances are if you have discovered the need for changes, someone else in the community needs them also.

10. **Development Environment Summary:** Aside from the previous steps, some best practices exist within the Yocto Project development environment. Consider the following:
- Use [Git](#) as the source control system.
 - Maintain your Metadata in layers that make sense for your situation. See the "[The Yocto Project Layer Model](#)" section in the Yocto Project Overview and Concepts Manual and the "[Understanding and Creating Layers](#)" section for more information on layers.
 - Separate the project's Metadata and code by using separate Git repositories. See the "[Yocto Project Source Repositories](#)" section in the Yocto Project Overview and Concepts Manual for information on these repositories. See the "[Locating Yocto Project Source Files](#)" section for information on how to set up local Git repositories for related upstream Yocto Project Git repositories.
 - Set up the directory for the shared state cache (`SSTATE_DIR`) where it makes sense. For example, set up the sstate cache on a system used by developers in the same organization and share the same source directories on their machines.
 - Set up an Autobuilder and have it populate the sstate cache and source directories.
 - The Yocto Project community encourages you to send patches to the project to fix bugs or add features. If you do submit patches, follow the project commit guidelines for writing good commit messages. See the "[Submitting a Change to the Yocto Project](#)" section.
 - Send changes to the core sooner than later as others are likely to run into the same issues. For some guidance on mailing lists to use, see the list in the "[Submitting a Change to the Yocto Project](#)" section. For a description of the available mailing lists,

see the "[Mailing Lists](#)" section in the Yocto Project Reference Manual.

6.2. Preparing the Build Host¶

This section provides procedures to set up a system to be used as your [build host](#) for development using the Yocto Project. Your build host can be a native Linux machine (recommended) or it can be a machine (Linux, Mac, or Windows) that uses [CROPS](#), which leverages [Docker Containers](#).

Note

You cannot use a build host that is using the [Windows Subsystem for Linux](#) (WSL). The Yocto Project is not compatible with WSL.

Once your build host is set up to use the Yocto Project, further steps are necessary depending on what you want to accomplish. See the following references for information on how to prepare for Board Support Package (BSP) development and kernel development:

- **BSP Development:** See the "[Preparing Your Build Host to Work With BSP Layers](#)" section in the Yocto Project Board Support Package (BSP) Developer's Guide.
- **Kernel Development:** See the "[Preparing the Build Host to Work on the Kernel](#)" section in the Yocto Project Linux Kernel Development Manual.

6.2.1. Setting Up a Native Linux Host¶

Follow these steps to prepare a native Linux machine as your Yocto Project Build Host:

1. **Use a Supported Linux Distribution:** You should have a reasonably current Linux-based host system. You will have the best results with a recent release of Fedora, openSUSE, Debian, Ubuntu, or CentOS as these releases are frequently tested against the Yocto Project and officially supported. For a list of the distributions under validation and their status, see the "[Supported Linux Distributions](#)" section in the Yocto Project Reference Manual and the wiki page at [Distribution Support](#).
2. **Have Enough Free Memory:** Your system should have at least 50 Gbytes of free disk space for building images.
3. **Meet Minimal Version Requirements:** The OpenEmbedded build system should be able to run on any modern distribution that has the following versions for Git, tar, and Python.

- Git 1.8.3.1 or greater
- tar 1.27 or greater
- Python 3.4.0 or greater.

If your build host does not meet any of these three listed version requirements, you can take steps to prepare the system so that you can still use the Yocto Project. See the "[Required Git, tar, and Python Versions](#)" section in the Yocto Project Reference Manual for information.

4. **Install Development Host Packages:** Required development host packages vary depending on your build host and what you want to do with the Yocto Project. Collectively, the number of required packages is large if you want to be able to cover all cases.

For lists of required packages for all scenarios, see the "[Required Packages for the Build Host](#)" section in the Yocto Project Reference Manual.

Once you have completed the previous steps, you are ready to continue using a given development path on your native Linux machine. If you are going to use BitBake, see the "[Cloning the Poky Repository](#)" section. If you are going to use the Extensible SDK, see the "[Using the Extensible SDK](#)" Chapter in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual. If you want to work on the kernel, see the Yocto Project Linux Kernel Development Manual. If you are going to use Toaster, see the "[Setting Up and Using Toaster](#)" section in the Toaster User Manual.

6.2.2. Setting Up to Use CROss Platforms (CROPS)¶

With [CROPS](#), which leverages [Docker Containers](#), you can create a Yocto Project development environment that is operating system agnostic. You can set up a container in which you can develop using the Yocto Project on a Windows, Mac, or Linux machine.

Follow these general steps to prepare a Windows, Mac, or Linux machine as your Yocto Project build host:

1. **Determine What Your Build Host Needs:** [Docker](#) is a software container platform that you need to install on the build host. Depending on your build host, you might have to install different software to support Docker containers. Go to the Docker installation page and read about the platform requirements in "[Supported Platforms](#)" your build host needs to run containers.
2. **Choose What To Install:** Depending on whether or not your build host meets system requirements, you need to install "Docker CE Stable" or the "Docker Toolbox". Most situations call for Docker CE. However, if you have a build host that does not meet requirements (e.g. Pre-Windows 10 or Windows 10 "Home" version), you must install Docker Toolbox instead.
3. **Go to the Install Site for Your Platform:** Click the link for the Docker edition associated with your build host's native software. For example, if your build host is running Microsoft Windows Version 10 and you want the Docker CE Stable edition, click that link under "Supported Platforms".
4. **Install the Software:** Once you have understood all the pre-requisites, you can download and install the appropriate software. Follow the instructions for your specific machine and the type of the software you need to install:
 - Install [Docker CE for Windows](#) for Windows build hosts that meet requirements.
 - Install [Docker CE for Macs](#) for Mac build hosts that meet requirements.
 - Install [Docker Toolbox for Windows](#) for Windows build hosts that do not meet Docker requirements.
 - Install [Docker Toolbox for MacOS](#) for Mac build hosts that do not meet Docker requirements.

- Install [Docker CE for CentOS](#) for Linux build hosts running the CentOS distribution.
 - Install [Docker CE for Debian](#) for Linux build hosts running the Debian distribution.
 - Install [Docker CE for Fedora](#) for Linux build hosts running the Fedora distribution.
 - Install [Docker CE for Ubuntu](#) for Linux build hosts running the Ubuntu distribution.
5. **Optionally Orient Yourself With Docker:** If you are unfamiliar with Docker and the container concept, you can learn more here - <https://docs.docker.com/get-started/>.
 6. **Launch Docker or Docker Toolbox:** You should be able to launch Docker or the Docker Toolbox and have a terminal shell on your development host.
 7. **Set Up the Containers to Use the Yocto Project:** Go to <https://github.com/crops/docker-win-mac-docs/wiki> and follow the directions for your particular build host (i.e. Linux, Mac, or Windows).

Once you complete the setup instructions for your machine, you have the Poky, Extensible SDK, and Toaster containers available. You can click those links from the page and learn more about using each of those containers.

Once you have a container set up, everything is in place to develop just as if you were running on a native Linux machine. If you are going to use the Poky container, see the "[Cloning the poky Repository](#)" section. If you are going to use the Extensible SDK container, see the "[Using the Extensible SDK](#)" Chapter in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual. If you are going to use the Toaster container, see the "[Setting Up and Using Toaster](#)" section in the Toaster User Manual.

6.3. Locating Yocto Project Source Files¶

This section shows you how to locate and access the source files that ship with the Yocto Project. You establish and use these local files to work on projects.

Notes

- For concepts and introductory information about Git as it is used in the Yocto Project, see the "[Git](#)" section in the Yocto Project Overview and Concepts Manual.
- For concepts on Yocto Project source repositories, see the "[Yocto Project Source Repositories](#)" section in the Yocto Project Overview and Concepts Manual."

6.3.1. Accessing Source Repositories¶

Working from a copy of the upstream Yocto Project [Source Repositories](#) is the preferred method for obtaining and using a Yocto Project release. You can view the Yocto Project Source Repositories at <http://git.yoctoproject.org>. In particular, you can find the poky repository at <http://git.yoctoproject.org/cgi/cgit.cgi/poky/>.

Use the following procedure to locate the latest upstream copy of the poky Git repository:

1. **Access Repositories:** Open a browser and go to <http://git.yoctoproject.org> to access the GUI-based interface into the Yocto Project source repositories.
2. **Select the Repository:** Click on the repository in which you are interested (e.g. poky).
3. **Find the URL Used to Clone the Repository:** At the bottom of the page, note the URL used to [clone](#) that repository (e.g. <http://git.yoctoproject.org/poky>).

Note

For information on cloning a repository, see the "[Cloning the poky Repository](#)" section.

6.3.2. Accessing Index of Releases¶

Yocto Project maintains an Index of Releases area that contains related files that contribute to the Yocto Project. Rather than Git repositories, these files are tarballs that represent snapshots in time of a given component.

Tip

The recommended method for accessing Yocto Project components is to use Git to clone the upstream repository and work from within that locally cloned repository. The procedure in this section exists should you desire a tarball snapshot of any given component.

Follow these steps to locate and download a particular tarball:

1. **Access the Index of Releases:** Open a browser and go to <http://downloads.yoctoproject.org/releases> to access the Index of Releases. The list represents released components (e.g. [bitbake](#), [sato](#), and so on).

Note

The `yocto` directory contains the full array of released Poky tarballs. The `poky` directory in the Index of Releases was historically used for very early releases and exists now only for retroactive

completeness.

2. **Select a Component:** Click on any released component in which you are interested (e.g. `yocto`).
3. **Find the Tarball:** Drill down to find the associated tarball. For example, click on `yocto-3.0.1` to view files associated with the Yocto Project 3.0.1 release (e.g. `poky-zeus-22.0.1.tar.bz2`, which is the released Poky tarball).
4. **Download the Tarball:** Click the tarball to download and save a snapshot of the given component.

6.3.3. Using the Downloads Page¶

The [Yocto Project Website](#) uses a "DOWNLOADS" page from which you can locate and download tarballs of any Yocto Project release. Rather than Git repositories, these files represent snapshot tarballs similar to the tarballs located in the Index of Releases described in the ["Accessing Index of Releases"](#) section.

Tip

The recommended method for accessing Yocto Project components is to use Git to clone a repository and work from within that local repository. The procedure in this section exists should you desire a tarball snapshot of any given component.

1. **Go to the Yocto Project Website:** Open The [Yocto Project Website](#) in your browser.
2. **Get to the Downloads Area:** Select the "DOWNLOADS" item from the pull-down "SOFTWARE" tab menu near the top of the page.
3. **Select a Yocto Project Release:** Use the menu next to "RELEASE" to display and choose a recent or past supported Yocto Project release (e.g. zeus, warrior, and so forth).

Tip

For a "map" of Yocto Project releases to version numbers, see the [Releases](#) wiki page.

You can use the "RELEASE ARCHIVE" link to reveal a menu of all Yocto Project releases.

4. **Download Tools or Board Support Packages (BSPs):** From the "DOWNLOADS" page, you can download tools or BSPs as well. Just scroll down the page and look for what you need.

6.3.4. Accessing Nightly Builds¶

Yocto Project maintains an area for nightly builds that contains tarball releases at <https://autobuilder.yocto.io/pub/nightly/>. These builds include Yocto Project releases ("poky"), toolchains, and builds for supported machines.

Should you ever want to access a nightly build of a particular Yocto Project component, use the following procedure:

1. **Locate the Index of Nightly Builds:** Open a browser and go to <https://autobuilder.yocto.io/pub/nightly/> to access the Nightly Builds.
2. **Select a Date:** Click on the date in which you are interested. If you want the latest builds, use "CURRENT".
3. **Select a Build:** Choose the area in which you are interested. For example, if you are looking for the most recent toolchains, select the "toolchain" link.
4. **Find the Tarball:** Drill down to find the associated tarball.
5. **Download the Tarball:** Click the tarball to download and save a snapshot of the given component.

6.4. Cloning and Checking Out Branches¶

To use the Yocto Project for development, you need a release locally installed on your development system. This locally installed set of files is referred to as the [Source Directory](#) in the Yocto Project documentation.

The preferred method of creating your Source Directory is by using [Git](#) to clone a local copy of the upstream `poky` repository. Working from a cloned copy of the upstream repository allows you to contribute back into the Yocto Project or to simply work with the latest software on a development branch. Because Git maintains and creates an upstream repository with a complete history of changes and you are working with a local clone of that repository, you have access to all the Yocto Project development branches and tag names used in the upstream repository.

6.4.1. Cloning the poky Repository¶

Follow these steps to create a local version of the upstream `poky` Git repository.

1. **Set Your Directory:** Change your working directory to where you want to create your local copy of `poky`.
2. **Clone the Repository:** The following example command clones the `poky` repository and uses the default name "poky" for your local repository:

```
$ git clone git://git.yoctoproject.org/poky
Cloning into 'poky'...
remote: Counting objects: 432160, done.
remote: Compressing objects: 100% (102056/102056), done.
remote: Total 432160 (delta 323116), reused 432037 (delta 323000)
```

```
Receiving objects: 100% (432160/432160), 153.81 MiB | 8.54 MiB/s, done.
Resolving deltas: 100% (323116/323116), done.
Checking connectivity... done.
```

Unless you specify a specific development branch or tag name, Git clones the "master" branch, which results in a snapshot of the latest development changes for "master". For information on how to check out a specific development branch or on how to check out a local branch based on a tag name, see the "[Checking Out By Branch in Poky](#)" and "[Checking Out By Tag in Poky](#)" sections, respectively.

Once the local repository is created, you can change to that directory and check its status. Here, the single "master" branch exists on your system and by default, it is checked out:

```
$ cd ~/poky
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
$ git branch
* master
```

Your local repository of poky is identical to the upstream poky repository at the time from which it was cloned. As you work with the local branch, you can periodically use the `git pull --rebase` command to be sure you are up-to-date with the upstream branch.

6.4.2. Checking Out by Branch in Poky¶

When you clone the upstream poky repository, you have access to all its development branches. Each development branch in a repository is unique as it forks off the "master" branch. To see and use the files of a particular development branch locally, you need to know the branch name and then specifically check out that development branch.

Note

Checking out an active development branch by branch name gives you a snapshot of that particular branch at the time you check it out. Further development on top of the branch that occurs after check it out can occur.

1. **Switch to the Poky Directory:** If you have a local poky Git repository, switch to that directory. If you do not have the local copy of poky, see the "[Cloning the poky Repository](#)" section.

2. **Determine Existing Branch Names:**

```
$ git branch -a
* master
remotes/origin/1.1_M1
remotes/origin/1.1_M2
remotes/origin/1.1_M3
remotes/origin/1.1_M4
remotes/origin/1.2_M1
remotes/origin/1.2_M2
remotes/origin/1.2_M3
.
.
.
remotes/origin/pyro
remotes/origin/pyro-next
remotes/origin/rocko
remotes/origin/rocko-next
remotes/origin/sumo
remotes/origin/sumo-next
remotes/origin/thud
remotes/origin/thud-next
remotes/origin/warrior
```

3. **Checkout the Branch:** Checkout the development branch in which you want to work. For example, to access the files for the Yocto Project 3.0.1 Release (Zeus), use the following command:

```
$ git checkout -b zeus origin/zeus
Branch zeus set up to track remote branch zeus from origin.
Switched to a new branch 'zeus'
```

The previous command checks out the "zeus" development branch and reports that the branch is tracking the upstream "origin/zeus" branch.

The following command displays the branches that are now part of your local poky repository. The asterisk character indicates the branch that is currently checked out for work:

```
$ git branch
master
* zeus
```

6.4.3. Checking Out by Tag in Poky¶

Similar to branches, the upstream repository uses tags to mark specific commits associated with significant points in a development branch (i.e. a release point or stage of a release). You might want to set up a local branch based on one of those points in the repository. The process is similar to checking out by branch name except you use tag names.

Note

Checking out a branch based on a tag gives you a stable set of files not affected by development on the branch above the tag.

1. **Switch to the Poky Directory:** If you have a local poky Git repository, switch to that directory. If you do not have the local copy of poky, see the "[Cloning the poky Repository](#)" section.
2. **Fetch the Tag Names:** To checkout the branch based on a tag name, you need to fetch the upstream tags into your local repository:

```
$ git fetch --tags
$
```

3. **List the Tag Names:** You can list the tag names now:

```
$ git tag
1.1_M1.final
1.1_M1.rc1
1.1_M1.rc2
1.1_M2.final
1.1_M2.rc1
.
.
.
yocto-2.5
yocto-2.5.1
yocto-2.5.2
yocto-2.5.3
yocto-2.6
yocto-2.6.1
yocto-2.6.2
yocto-2.7
yocto_1.5_M5.rc8
```

4. **Checkout the Branch:**

```
$ git checkout tags/yocto-3.0.1 -b my_yocto_3.0.1
Switched to a new branch 'my_yocto_3.0.1'
$ git branch
  master
* my_yocto_3.0.1
```

The previous command creates and checks out a local branch named "my_yocto_3.0.1", which is based on the commit in the upstream poky repository that has the same tag. In this example, the files you have available locally as a result of the `checkout` command are a snapshot of the "zeus" development branch at the point where Yocto Project 3.0.1 was released.

Chapter 7. Common Tasks¶

7.1. Understanding and Creating Layers

- [7.1.1. Creating Your Own Layer](#)
- [7.1.2. Following Best Practices When Creating Layers](#)
- [7.1.3. Making Sure Your Layer is Compatible With Yocto Project](#)
- [7.1.4. Enabling Your Layer](#)
- [7.1.5. Using .bbappend Files in Your Layer](#)
- [7.1.6. Prioritizing Your Layer](#)
- [7.1.7. Managing Layers](#)
- [7.1.8. Creating a General Layer Using the `bitbake-layers` Script](#)
- [7.1.9. Adding a Layer Using the `bitbake-layers` Script](#)

7.2. Customizing Images

- [7.2.1. Customizing Images Using `local.conf`](#)
- [7.2.2. Customizing Images Using Custom `IMAGE\_FEATURES` and `EXTRA\_IMAGE\_FEATURES`](#)
- [7.2.3. Customizing Images Using Custom .bb Files](#)
- [7.2.4. Customizing Images Using Custom Package Groups](#)
- [7.2.5. Customizing an Image Hostname](#)

7.3. Writing a New Recipe

- [7.3.1. Overview](#)
- [7.3.2. Locate or Automatically Create a Base Recipe](#)
- [7.3.3. Storing and Naming the Recipe](#)
- [7.3.4. Running a Build on the Recipe](#)
- [7.3.5. Fetching Code](#)
- [7.3.6. Unpacking Code](#)
- [7.3.7. Patching Code](#)
- [7.3.8. Licensing](#)
- [7.3.9. Dependencies](#)
- [7.3.10. Configuring the Recipe](#)
- [7.3.11. Using Headers to Interface with Devices](#)
- [7.3.12. Compilation](#)
- [7.3.13. Installing](#)
- [7.3.14. Enabling System Services](#)
- [7.3.15. Packaging](#)
- [7.3.16. Sharing Files Between Recipes](#)
- [7.3.17. Using Virtual Providers](#)
- [7.3.18. Properly Versioning Pre-Release Recipes](#)
- [7.3.19. Post-Installation Scripts](#)
- [7.3.20. Testing](#)
- [7.3.21. Examples](#)
- [7.3.22. Following Recipe Style Guidelines](#)
- [7.3.23. Recipe Syntax](#)

7.4. Adding a New Machine

- [7.4.1. Adding the Machine Configuration File](#)
- [7.4.2. Adding a Kernel for the Machine](#)
- [7.4.3. Adding a Formfactor Configuration File](#)

- [7.5. Upgrading Recipes](#)
 - [7.5.1. Using the Auto Upgrade Helper \(AUH\)](#)
 - [7.5.2. Using `devtool upgrade`](#)
 - [7.5.3. Manually Upgrading a Recipe](#)
- [7.6. Finding Temporary Source Code](#)
- [7.7. Using Quilt in Your Workflow](#)
- [7.8. Using a Development Shell](#)
- [7.9. Using a Development Python Shell](#)
- [7.10. Building](#)
 - [7.10.1. Building a Simple Image](#)
 - [7.10.2. Building Images for Multiple Targets Using Multiple Configurations](#)
 - [7.10.3. Building an Initial RAM Filesystem \(initramfs\) Image](#)
 - [7.10.4. Building a Tiny System](#)
 - [7.10.5. Building Images for More than One Machine](#)
 - [7.10.6. Building Software from an External Source](#)
 - [7.10.7. Replicating a Build Offline](#)
- [7.11. Speeding Up a Build](#)
- [7.12. Working With Libraries](#)
 - [7.12.1. Including Static Library Files](#)
 - [7.12.2. Combining Multiple Versions of Library Files into One Image](#)
 - [7.12.3. Installing Multiple Versions of the Same Library](#)
- [7.13. Using `x32 psABI`](#)
- [7.14. Enabling GObject Introspection Support](#)
 - [7.14.1. Enabling the Generation of Introspection Data](#)
 - [7.14.2. Disabling the Generation of Introspection Data](#)
 - [7.14.3. Testing that Introspection Works in an Image](#)
 - [7.14.4. Known Issues](#)
- [7.15. Optionally Using an External Toolchain](#)
- [7.16. Creating Partitioned Images Using Wic](#)
 - [7.16.1. Background](#)
 - [7.16.2. Requirements](#)
 - [7.16.3. Getting Help](#)
 - [7.16.4. Operational Modes](#)
 - [7.16.5. Using an Existing Kickstart File](#)
 - [7.16.6. Using the Wic Plugin Interface](#)
 - [7.16.7. Examples](#)
- [7.17. Flashing Images Using `bmaptool`](#)
- [7.18. Making Images More Secure](#)
 - [7.18.1. General Considerations](#)
 - [7.18.2. Security Flags](#)
 - [7.18.3. Considerations Specific to the OpenEmbedded Build System](#)
 - [7.18.4. Tools for Hardening Your Image](#)
- [7.19. Creating Your Own Distribution](#)
- [7.20. Creating a Custom Template Configuration Directory](#)
- [7.21. Conserving Disk Space During Builds](#)
- [7.22. Working with Packages](#)
 - [7.22.1. Excluding Packages from an Image](#)
 - [7.22.2. Incrementing a Package Version](#)
 - [7.22.3. Handling Optional Module Packaging](#)
 - [7.22.4. Using Runtime Package Management](#)
 - [7.22.5. Generating and Using Signed Packages](#)
 - [7.22.6. Testing Packages With `ptest`](#)
 - [7.22.7. Creating Node Package Manager \(NPM\) Packages](#)
- [7.23. Efficiently Fetching Source Files During a Build](#)
 - [7.23.1. Setting up Effective Mirrors](#)
 - [7.23.2. Getting Source Files and Suppressing the Build](#)
- [7.24. Selecting an Initialization Manager](#)
 - [7.24.1. Using `systemd` Exclusively](#)
 - [7.24.2. Using `systemd` for the Main Image and Using `SysVinit` for the Rescue Image](#)
- [7.25. Selecting a Device Manager](#)
 - [7.25.1. Using Persistent and Pre-Populated `/dev`](#)
 - [7.25.2. Using `devtmpfs` and a Device Manager](#)
- [7.26. Using an External SCM](#)
- [7.27. Creating a Read-Only Root Filesystem](#)
 - [7.27.1. Creating the Root Filesystem](#)
 - [7.27.2. Post-Installation Scripts](#)
 - [7.27.3. Areas With Write Access](#)
- [7.28. Maintaining Build Output Quality](#)
 - [7.28.1. Enabling and Disabling Build History](#)
 - [7.28.2. Understanding What the Build History Contains](#)
- [7.29. Performing Automated Runtime Testing](#)
 - [7.29.1. Enabling Tests](#)
 - [7.29.2. Running Tests](#)
 - [7.29.3. Exporting Tests](#)
 - [7.29.4. Writing New Tests](#)
 - [7.29.5. Installing Packages in the DUT Without the Package Manager](#)
- [7.30. Debugging Tools and Techniques](#)
 - [7.30.1. Viewing Logs from Failed Tasks](#)
 - [7.30.2. Viewing Variable Values](#)
 - [7.30.3. Viewing Package Information with `oe-pkgdata-util`](#)
 - [7.30.4. Viewing Dependencies Between Recipes and Tasks](#)
 - [7.30.5. Viewing Task Variable Dependencies](#)
 - [7.30.6. Viewing Metadata Used to Create the Input Signature of a Shared State Task](#)
 - [7.30.7. Invalidating Shared State to Force a Task to Run](#)
 - [7.30.8. Running Specific Tasks](#)
 - [7.30.9. General BitBake Problems](#)
 - [7.30.10. Building with No Dependencies](#)
 - [7.30.11. Recipe Logging Mechanisms](#)
 - [7.30.12. Debugging Parallel Make Races](#)
 - [7.30.13. Debugging With the GNU Project Debugger \(GDB\) Remotely](#)
 - [7.30.14. Debugging with the GNU Project Debugger \(GDB\) on the Target](#)
 - [7.30.15. Other Debugging Tips](#)
- [7.31. Making Changes to the Yocto Project](#)
 - [7.31.1. Submitting a Defect Against the Yocto Project](#)
 - [7.31.2. Submitting a Change to the Yocto Project](#)

- 7.32. [Working With Licenses](#)
 - 7.32.1. [Tracking License Changes](#)
 - 7.32.2. [Enabling Commercially Licensed Recipes](#)
 - 7.32.3. [Maintaining Open Source License Compliance During Your Product's Lifecycle](#)
 - 7.32.4. [Copying Licenses that Do Not Exist](#)
- 7.33. [Using the Error Reporting Tool](#)
 - 7.33.1. [Enabling and Using the Tool](#)
 - 7.33.2. [Disabling the Tool](#)
 - 7.33.3. [Setting Up Your Own Error Reporting Server](#)
- 7.34. [Using Wayland and Weston](#)
 - 7.34.1. [Enabling Wayland in an Image](#)
 - 7.34.2. [Running Weston](#)

This chapter describes fundamental procedures such as creating layers, adding new software packages, extending or customizing images, porting work to new hardware (adding a new machine), and so forth. You will find that the procedures documented here occur often in the development cycle using the Yocto Project.

7.1. Understanding and Creating Layers¶

The OpenEmbedded build system supports organizing [Metadata](#) into multiple layers. Layers allow you to isolate different types of customizations from each other. For introductory information on the Yocto Project Layer Model, see the "[The Yocto Project Layer Model](#)" section in the Yocto Project Overview and Concepts Manual.

7.1.1. Creating Your Own Layer¶

It is very easy to create your own layers to use with the OpenEmbedded build system. The Yocto Project ships with tools that speed up creating layers. This section describes the steps you perform by hand to create layers so that you can better understand them. For information about the layer-creation tools, see the "[Creating a New BSP Layer Using the `bitbake-layers` Script](#)" section in the Yocto Project Board Support Package (BSP) Developer's Guide and the "[Creating a General Layer Using the `bitbake-layers` Script](#)" section further down in this manual.

Follow these general steps to create your layer without using tools:

1. **Check Existing Layers:** Before creating a new layer, you should be sure someone has not already created a layer containing the Metadata you need. You can see the [OpenEmbedded Metadata Index](#) for a list of layers from the OpenEmbedded community that can be used in the Yocto Project. You could find a layer that is identical or close to what you need.
2. **Create a Directory:** Create the directory for your layer. When you create the layer, be sure to create the directory in an area not associated with the Yocto Project [Source Directory](#). (e.g. the cloned `poky` repository).

While not strictly required, prepend the name of the directory with the string "meta-". For example:

```
meta-mylayer
meta-GUI_xyz
meta-mymachine
```

With rare exceptions, a layer's name follows this form:

```
meta-root_name
```

Following this layer naming convention can save you trouble later when tools, components, or variables "assume" your layer name begins with "meta-". A notable example is in configuration files as shown in the following step where layer names without the "meta-" string are appended to several variables used in the configuration.

3. **Create a Layer Configuration File:** Inside your new layer folder, you need to create a `conf/layer.conf` file. It is easiest to take an existing layer configuration file and copy that to your layer's `conf` directory and then modify the file as needed.

The `meta-yocto-bsp/conf/layer.conf` file in the Yocto Project [Source Repositories](#) demonstrates the required syntax. For your layer, you need to replace "yoctobsp" with a unique identifier for your layer (e.g. "machinexyz" for a layer named "meta-machinexyz"):

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

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

BBFILE_COLLECTIONS += "yoctobsp"
BBFILE_PATTERN_yoctobsp = "^${LAYERDIR}/"
BBFILE_PRIORITY_yoctobsp = "5"
LAYERVERSION_yoctobsp = "4"
LAYERSERIES_COMPAT_yoctobsp = "zeus"
```

Following is an explanation of the layer configuration file:

- **[BBPATH](#):** Adds the layer's root directory to BitBake's search path. Through the use of the `BBPATH` variable, BitBake locates class files (`.bbclass`), configuration files, and files that are included with `include` and `require` statements. For these cases, BitBake uses the first file that matches the name found in `BBPATH`. This is similar to the way the `PATH` variable is used for binaries. It is recommended, therefore, that you use unique class and configuration filenames in your custom layer.
- **[BBFILES](#):** Defines the location for all recipes in the layer.
- **[BBFILE_COLLECTIONS](#):** Establishes the current layer through a unique identifier that is used throughout the OpenEmbedded build system to refer to the layer. In this example, the identifier "yoctobsp" is the representation for the container layer named "meta-yocto-bsp".

- **BBFILE_PATTERN**: Expands immediately during parsing to provide the directory of the layer.
 - **BBFILE_PRIORITY**: Establishes a priority to use for recipes in the layer when the OpenEmbedded build finds recipes of the same name in different layers.
 - **LAYERVERSION**: Establishes a version number for the layer. You can use this version number to specify this exact version of the layer as a dependency when using the **LAYERDEPENDS** variable.
 - **LAYERSERIES_COMPAT**: Lists the [Yocto Project](#) releases for which the current version is compatible. This variable is a good way to indicate if your particular layer is current.
4. **Add Content**: Depending on the type of layer, add the content. If the layer adds support for a machine, add the machine configuration in a `conf/machine/` file within the layer. If the layer adds distro policy, add the distro configuration in a `conf/distro/` file within the layer. If the layer introduces new recipes, put the recipes you need in `recipes-*` subdirectories within the layer.

Note

For an explanation of layer hierarchy that is compliant with the Yocto Project, see the "[Example Filesystem Layout](#)" section in the Yocto Project Board Support Package (BSP) Developer's Guide.

5. **Optionally Test for Compatibility**: If you want permission to use the Yocto Project Compatibility logo with your layer or application that uses your layer, perform the steps to apply for compatibility. See the "[Making Sure Your Layer is Compatible With Yocto Project](#)" section for more information.

7.1.2. Following Best Practices When Creating Layers¶

To create layers that are easier to maintain and that will not impact builds for other machines, you should consider the information in the following list:

- **Avoid "Overlaying" Entire Recipes from Other Layers in Your Configuration**: In other words, do not copy an entire recipe into your layer and then modify it. Rather, use an append file (`.bbappend`) to override only those parts of the original recipe you need to modify.
- **Avoid Duplicating Include Files**: Use append files (`.bbappend`) for each recipe that uses an include file. Or, if you are introducing a new recipe that requires the included file, use the path relative to the original layer directory to refer to the file. For example, use `require recipes-core/package/file.inc` instead of `require file.inc`. If you're finding you have to overlay the include file, it could indicate a deficiency in the include file in the layer to which it originally belongs. If this is the case, you should try to address that deficiency instead of overlaying the include file. For example, you could address this by getting the maintainer of the include file to add a variable or variables to make it easy to override the parts needing to be overridden.
- **Structure Your Layers**: Proper use of overrides within append files and placement of machine-specific files within your layer can ensure that a build is not using the wrong Metadata and negatively impacting a build for a different machine. Following are some examples:
 - **Modify Variables to Support a Different Machine**: Suppose you have a layer named `meta-one` that adds support for building machine "one". To do so, you use an append file named `base-files.bbappend` and create a dependency on "foo" by altering the **DEPENDS** variable:

```
DEPENDS = "foo"
```

The dependency is created during any build that includes the layer `meta-one`. However, you might not want this dependency for all machines. For example, suppose you are building for machine "two" but your `bblayers.conf` file has the `meta-one` layer included. During the build, the `base-files` for machine "two" will also have the dependency on `foo`.

To make sure your changes apply only when building machine "one", use a machine override with the **DEPENDS** statement:

```
DEPENDS_one = "foo"
```

You should follow the same strategy when using `_append` and `_prepend` operations:

```
DEPENDS_append_one = " foo"
DEPENDS_prepend_one = "foo "
```

As an actual example, here's a line from the recipe for `gnutls`, which adds dependencies on "argp-standalone" when building with the `musl` C library:

```
DEPENDS_append_libc-musl = " argp-standalone"
```

Note

Avoiding "+=" and "=+" and using machine-specific `_append` and `_prepend` operations is recommended as well.

- **Place Machine-Specific Files in Machine-Specific Locations**: When you have a base recipe, such as `base-files.bb`, that contains a **SRC_URI** statement to a file, you can use an append file to cause the build to use your own version of the file. For example, an append file in your layer at `meta-one/recipes-core/base-files/base-files.bbappend` could extend **FILESPATH** using **FILESEXTRAPATHS** as follows:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${BPN}:"
```

The build for machine "one" will pick up your machine-specific file as long as you have the file in `meta-one/recipes-core/base-files/base-files/`. However, if you are building for a different machine and the `bblayers.conf` file includes the `meta-one` layer and the location of your machine-specific file is the first location where that file is found according to `FILESPATH`, builds for all machines will also use that machine-specific file.

You can make sure that a machine-specific file is used for a particular machine by putting the file in a subdirectory specific to the machine. For example, rather than placing the file in `meta-one/recipes-core/base-files/base-files/` as shown above, put it in `meta-one/recipes-core/base-files/base-files/one/`. Not only does this make sure the file is used only when building for machine "one", but the build process locates the file more quickly.

In summary, you need to place all files referenced from `SRC_URI` in a machine-specific subdirectory within the layer in order to restrict those files to machine-specific builds.

- **Perform Steps to Apply for Yocto Project Compatibility:** If you want permission to use the Yocto Project Compatibility logo with your layer or application that uses your layer, perform the steps to apply for compatibility. See the "[Making Sure Your Layer is Compatible With Yocto Project](#)" section for more information.
- **Follow the Layer Naming Convention:** Store custom layers in a Git repository that use the `meta-layer_name` format.
- **Group Your Layers Locally:** Clone your repository alongside other cloned `meta` directories from the [Source Directory](#).

7.1.3. Making Sure Your Layer is Compatible With Yocto Project¶

When you create a layer used with the Yocto Project, it is advantageous to make sure that the layer interacts well with existing Yocto Project layers (i.e. the layer is compatible with the Yocto Project). Ensuring compatibility makes the layer easy to be consumed by others in the Yocto Project community and could allow you permission to use the Yocto Project Compatible Logo.

Note

Only Yocto Project member organizations are permitted to use the Yocto Project Compatible Logo. The logo is not available for general use. For information on how to become a Yocto Project member organization, see the [Yocto Project Website](#).

The Yocto Project Compatibility Program consists of a layer application process that requests permission to use the Yocto Project Compatibility Logo for your layer and application. The process consists of two parts:

1. Successfully passing a script (`yocto-check-layer`) that when run against your layer, tests it against constraints based on experiences of how layers have worked in the real world and where pitfalls have been found. Getting a "PASS" result from the script is required for successful compatibility registration.
2. Completion of an application acceptance form, which you can find at <https://www.yoctoproject.org/webform/yocto-project-compatible-registration>.

To be granted permission to use the logo, you need to satisfy the following:

- Be able to check the box indicating that you got a "PASS" when running the script against your layer.
- Answer "Yes" to the questions on the form or have an acceptable explanation for any questions answered "No".
- Be a Yocto Project Member Organization.

The remainder of this section presents information on the registration form and on the `yocto-check-layer` script.

7.1.3.1. Yocto Project Compatible Program Application¶

Use the form to apply for your layer's approval. Upon successful application, you can use the Yocto Project Compatibility Logo with your layer and the application that uses your layer.

To access the form, use this link: <https://www.yoctoproject.org/webform/yocto-project-compatible-registration>. Follow the instructions on the form to complete your application.

The application consists of the following sections:

- **Contact Information:** Provide your contact information as the fields require. Along with your information, provide the released versions of the Yocto Project for which your layer is compatible.
- **Acceptance Criteria:** Provide "Yes" or "No" answers for each of the items in the checklist. Space exists at the bottom of the form for any explanations for items for which you answered "No".
- **Recommendations:** Provide answers for the questions regarding Linux kernel use and build success.

7.1.3.2. yocto-check-layer Script¶

The `yocto-check-layer` script provides you a way to assess how compatible your layer is with the Yocto Project. You should run this script prior to using the form to apply for compatibility as described in the previous section. You need to achieve a "PASS" result in order to have your application form successfully processed.

The script divides tests into three areas: COMMON, BSP, and DISTRO. For example, given a distribution layer (DISTRO), the layer must pass both the COMMON and DISTRO related tests. Furthermore, if your layer is a BSP layer, the layer must pass the COMMON and BSP set of tests.

To execute the script, enter the following commands from your build directory:

```
$ source oe-init-build-env
$ yocto-check-layer your_layer_directory
```

Be sure to provide the actual directory for your layer as part of the command.

Entering the command causes the script to determine the type of layer and then to execute a set of specific tests against the layer. The following list overviews the test:

- `common.test_readme`: Tests if a `README` file exists in the layer and the file is not empty.
- `common.test_parse`: Tests to make sure that BitBake can parse the files without error (i.e. `bitbake -p`).
- `common.test_show_environment`: Tests that the global or per-recipe environment is in order without errors (i.e. `bitbake -e`).
- `common.test_signatures`: Tests to be sure that BSP and DISTRO layers do not come with recipes that change signatures.
- `bsp.test_bsp_defines_machines`: Tests if a BSP layer has machine configurations.
- `bsp.test_bsp_no_set_machine`: Tests to ensure a BSP layer does not set the machine when the layer is added.
- `distro.test_distro_defines_distros`: Tests if a DISTRO layer has distro configurations.
- `distro.test_distro_no_set_distro`: Tests to ensure a DISTRO layer does not set the distribution when the layer is added.

7.1.4. Enabling Your Layer¶

Before the OpenEmbedded build system can use your new layer, you need to enable it. To enable your layer, simply add your layer's path to the `BBLAYERS` variable in your `conf/bblayers.conf` file, which is found in the [Build Directory](#). The following example shows how to enable a layer named `meta-mylayer`:

```
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

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

BBLAYERS ?= " \
    /home/user/poky/meta \
    /home/user/poky/meta-poky \
    /home/user/poky/meta-yocto-bsp \
    /home/user/poky/meta-mylayer \
"
```

BitBake parses each `conf/layer.conf` file from the top down as specified in the `BBLAYERS` variable within the `conf/bblayers.conf` file. During the processing of each `conf/layer.conf` file, BitBake adds the recipes, classes and configurations contained within the particular layer to the source directory.

7.1.5. Using .bbappend Files in Your Layer¶

A recipe that appends Metadata to another recipe is called a BitBake append file. A BitBake append file uses the `.bbappend` file type suffix, while the corresponding recipe to which Metadata is being appended uses the `.bb` file type suffix.

You can use a `.bbappend` file in your layer to make additions or changes to the content of another layer's recipe without having to copy the other layer's recipe into your layer. Your `.bbappend` file resides in your layer, while the main `.bb` recipe file to which you are appending Metadata resides in a different layer.

Being able to append information to an existing recipe not only avoids duplication, but also automatically applies recipe changes from a different layer into your layer. If you were copying recipes, you would have to manually merge changes as they occur.

When you create an append file, you must use the same root name as the corresponding recipe file. For example, the append file `someapp_3.0.1.bbappend` must apply to `someapp_3.0.1.bb`. This means the original recipe and append file names are version number-specific. If the corresponding recipe is renamed to update to a newer version, you must also rename and possibly update the corresponding `.bbappend` as well. During the build process, BitBake displays an error on starting if it detects a `.bbappend` file that does not have a corresponding recipe with a matching name. See the [BB_DANGLINGAPPENDS_WARNONLY](#) variable for information on how to handle this error.

As an example, consider the main formfactor recipe and a corresponding formfactor append file both from the [Source Directory](#). Here is the main formfactor recipe, which is named `formfactor_0.0.bb` and located in the "meta" layer at `meta/recipes-bsp/formfactor`:

```
SUMMARY = "Device formfactor information"
SECTION = "base"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COREBASE}/meta/COPYING.MIT;md5=3da9cfbcb788c80a0384361b4de20420"
PR = "r45"

SRC_URI = "file://config file://machconfig"
S = "${WORKDIR}"

PACKAGE_ARCH = "${MACHINE_ARCH}"
INHIBIT_DEFAULT_DEPS = "1"

do_install() {
    # Install file only if it has contents
    install -d ${D}${sysconfdir}/formfactor/
    install -m 0644 ${S}/config ${D}${sysconfdir}/formfactor/
    if [ -s "${S}/machconfig" ]; then
        install -m 0644 ${S}/machconfig ${D}${sysconfdir}/formfactor/
    fi
}
```

In the main recipe, note the `SRC_URI` variable, which tells the OpenEmbedded build system where to find files during the build.

Following is the append file, which is named `formfactor_0.0.bbappend` and is from the Raspberry Pi BSP Layer named `meta-raspberrypi`. The file is in the layer at `recipes-bsp/formfactor`:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
```

By default, the build system uses the `FILESPATH` variable to locate files. This append file extends the locations by setting the `FILESEXTRAPATHS` variable. Setting this variable in the `.bbappend` file is the most reliable and recommended method for adding directories to the search path used by the build system to find files.

The statement in this example extends the directories to include `${THISDIR}/${PN}`, which resolves to a directory named `formfactor` in the same directory in which the append file resides (i.e. `meta-raspberrypi/recipes-bsp/formfactor`). This implies that you must have the supporting directory structure set up that will contain any files or patches you will be including from the layer.

Using the immediate expansion assignment operator `:=` is important because of the reference to `THISDIR`. The trailing colon character is important as it ensures that items in the list remain colon-separated.

Note

BitBake automatically defines the `THISDIR` variable. You should never set this variable yourself. Using `"_prepend"` as part of the `FILESEXTRAPATHS` ensures your path will be searched prior to other paths in the final list.

Also, not all append files add extra files. Many append files simply exist to add build options (e.g. `systemd`). For these cases, your append file would not even use the `FILESEXTRAPATHS` statement.

7.1.6. Prioritizing Your Layer¶

Each layer is assigned a priority value. Priority values control which layer takes precedence if there are recipe files with the same name in multiple layers. For these cases, the recipe file from the layer with a higher priority number takes precedence. Priority values also affect the order in which multiple `.bbappend` files for the same recipe are applied. You can either specify the priority manually, or allow the build system to calculate it based on the layer's dependencies.

To specify the layer's priority manually, use the `BBFILE_PRIORITY` variable and append the layer's root name:

```
BBFILE_PRIORITY_mylayer = "1"
```

Note

It is possible for a recipe with a lower version number `PV` in a layer that has a higher priority to take precedence.

Also, the layer priority does not currently affect the precedence order of `.conf` or `.bbclass` files. Future versions of BitBake might address this.

7.1.7. Managing Layers¶

You can use the BitBake layer management tool `bitbake-layers` to provide a view into the structure of recipes across a multi-layer project. Being able to generate output that reports on configured layers with their paths and priorities and on `.bbappend` files and their applicable recipes can help to reveal potential problems.

For help on the BitBake layer management tool, use the following command:

```
$ bitbake-layers --help
NOTE: Starting bitbake server...
usage: bitbake-layers [-d] [-q] [-F] [--color COLOR] [-h] <subcommand> ...

BitBake layers utility

optional arguments:
  -d, --debug          Enable debug output
  -q, --quiet          Print only errors
  -F, --force          Force add without recipe parse verification
  --color COLOR        Colorize output (where COLOR is auto, always, never)
  -h, --help          show this help message and exit

subcommands:
  <subcommand>
  show-layers          show current configured layers.
  show-overlaid        list overlaid recipes (where the same recipe exists
                        in another layer)
  show-recipes         list available recipes, showing the layer they are
                        provided by
  show-appends         list bbappend files and recipe files they apply to
  show-cross-depends  Show dependencies between recipes that cross layer
                        boundaries.
  add-layer            Add one or more layers to bblayers.conf.
  remove-layer         Remove one or more layers from bblayers.conf.
  flatten              flatten layer configuration into a separate output
                        directory.
  layerindex-fetch     Fetches a layer from a layer index along with its
                        dependent layers, and adds them to conf/bblayers.conf.
  layerindex-show-depends
                        Find layer dependencies from layer index.
  create-layer         Create a basic layer
```

Use `bitbake-layers <subcommand> --help` to get help on a specific command

The following list describes the available commands:

- **help**: Displays general help or help on a specified command.
- **show-layers**: Shows the current configured layers.
- **show-overlayed**: Lists overlayed recipes. A recipe is overlayed when a recipe with the same name exists in another layer that has a higher layer priority.
- **show-recipes**: Lists available recipes and the layers that provide them.
- **show-appends**: Lists `.bbappend` files and the recipe files to which they apply.
- **show-cross-depends**: Lists dependency relationships between recipes that cross layer boundaries.
- **add-layer**: Adds a layer to `bblayers.conf`.
- **remove-layer**: Removes a layer from `bblayers.conf`.
- **flatten**: Flattens the layer configuration into a separate output directory. Flattening your layer configuration builds a "flattened" directory that contains the contents of all layers, with any overlayed recipes removed and any `.bbappend` files appended to the corresponding recipes. You might have to perform some manual cleanup of the flattened layer as follows:
 - Non-recipe files (such as patches) are overwritten. The `flatten` command shows a warning for these files.
 - Anything beyond the normal layer setup has been added to the `layer.conf` file. Only the lowest priority layer's `layer.conf` is used.
 - Overridden and appended items from `.bbappend` files need to be cleaned up. The contents of each `.bbappend` end up in the flattened recipe. However, if there are appended or changed variable values, you need to tidy these up yourself. Consider the following example. Here, the `bitbake-layers` command adds the line `#### bbappended . . .` so that you know where the following lines originate:

```
...
DESCRIPTION = "A useful utility"
...
EXTRA_OECONF = "--enable-something"
...

#### bbappended from meta-anotherlayer ####

DESCRIPTION = "Customized utility"
EXTRA_OECONF += "--enable-somethingelse"
```

Ideally, you would tidy up these utilities as follows:

```
...
DESCRIPTION = "Customized utility"
...
EXTRA_OECONF = "--enable-something --enable-somethingelse"
...
```

- **layerindex-fetch**: Fetches a layer from a layer index, along with its dependent layers, and adds the layers to the `conf/bblayers.conf` file.
- **layerindex-show-depends**: Finds layer dependencies from the layer index.
- **create-layer**: Creates a basic layer.

7.1.8. Creating a General Layer Using the `bitbake-layers` Script

The `bitbake-layers` script with the `create-layer` subcommand simplifies creating a new general layer.

Notes

- For information on BSP layers, see the "[BSP Layers](#)" section in the Yocto Project Board Specific (BSP) Developer's Guide.
- In order to use a layer with the OpenEmbedded build system, you need to add the layer to your `bblayers.conf` configuration file. See the "[Adding a Layer Using the `bitbake-layers` Script](#)" section for more information.

The default mode of the script's operation with this subcommand is to create a layer with the following:

- A layer priority of 6.
- A `conf` subdirectory that contains a `layer.conf` file.
- A `recipes-example` subdirectory that contains a further subdirectory named `example`, which contains an `example.bb` recipe file.
- A `COPYING.MIT`, which is the license statement for the layer. The script assumes you want to use the MIT license, which is typical for most layers, for the contents of the layer itself.
- A `README` file, which is a file describing the contents of your new layer.

In its simplest form, you can use the following command form to create a layer. The command creates a layer whose name corresponds to `your_layer_name` in the current directory:

```
$ bitbake-layers create-layer your_Layer_name
```

As an example, the following command creates a layer named `meta-scottrif` in your home directory:

```
$ cd /usr/home
$ bitbake-layers create-layer meta-scottrif
NOTE: Starting bitbake server...
Add your new layer with 'bitbake-layers add-layer meta-scottrif'
```

If you want to set the priority of the layer to other than the default value of "6", you can either use the `--priority` option or you can edit the `BBFILE_PRIORITY` value in the `conf/layer.conf` after the script creates it. Furthermore, if you want to give the example recipe file some name other than the default, you can use the `--example-recipe-name` option.

The easiest way to see how the `bitbake-layers create-layer` command works is to experiment with the script. You can also read the usage information by entering the following:

```
$ bitbake-layers create-layer --help
NOTE: Starting bitbake server...
usage: bitbake-layers create-layer [-h] [--priority PRIORITY]
                                   [--example-recipe-name EXAMPLEREcipe]
                                   layerdir
```

Create a basic layer

positional arguments:

layerdir Layer directory to create

optional arguments:

```
-h, --help show this help message and exit
--priority PRIORITY, -p PRIORITY
            Layer directory to create
--example-recipe-name EXAMPLEREcipe, -e EXAMPLEREcipe
            Filename of the example recipe
```

7.1.9. Adding a Layer Using the `bitbake-layers` Script¶

Once you create your general layer, you must add it to your `bblayers.conf` file. Adding the layer to this configuration file makes the OpenEmbedded build system aware of your layer so that it can search it for metadata.

Add your layer by using the `bitbake-layers add-layer` command:

```
$ bitbake-layers add-layer your_Layer_name
```

Here is an example that adds a layer named `meta-scottrif` to the configuration file. Following the command that adds the layer is another `bitbake-layers` command that shows the layers that are in your `bblayers.conf` file:

```
$ bitbake-layers add-layer meta-scottrif
NOTE: Starting bitbake server...
Parsing recipes: 100% |#####| Time: 0:00:49
Parsing of 1441 .bb files complete (0 cached, 1441 parsed). 2055 targets, 56 skipped, 0 masked, 0 errors.
$ bitbake-layers show-layers
NOTE: Starting bitbake server...
layer path priority
=====
meta /home/scottrif/poky/meta 5
meta-poky /home/scottrif/poky/meta-poky 5
meta-yocto-bsp /home/scottrif/poky/meta-yocto-bsp 5
workspace /home/scottrif/poky/build/workspace 99
meta-scottrif /home/scottrif/poky/build/meta-scottrif 6
```

Adding the layer to this file enables the build system to locate the layer during the build.

Note

During a build, the OpenEmbedded build system looks in the layers from the top of the list down to the bottom in that order.

7.2. Customizing Images¶

You can customize images to satisfy particular requirements. This section describes several methods and provides guidelines for each.

7.2.1. Customizing Images Using `local.conf`¶

Probably the easiest way to customize an image is to add a package by way of the `local.conf` configuration file. Because it is limited to local use, this method generally only allows you to add packages and is not as flexible as creating your own customized image. When you add packages using local variables this way, you need to realize that these variable changes are in effect for every build and consequently affect all images, which might not be what you require.

To add a package to your image using the local configuration file, use the `IMAGE_INSTALL` variable with the `_append` operator:

```
IMAGE_INSTALL_append = " strace"
```

Use of the syntax is important - specifically, the space between the quote and the package name, which is `strace` in this example. This space is required since the `_append` operator does not add the space.

Furthermore, you must use `_append` instead of the `+=` operator if you want to avoid ordering issues. The reason for this is because doing so unconditionally appends to the variable and avoids ordering problems due to the variable being set in image recipes and `.bbclass` files with operators like `?=`. Using `_append` ensures the operation takes affect.

As shown in its simplest use, `IMAGE_INSTALL_append` affects all images. It is possible to extend the syntax so that the variable applies to a specific image only. Here is an example:

```
IMAGE_INSTALL_append_pn-core-image-minimal = " strace"
```

This example adds `strace` to the `core-image-minimal` image only.

You can add packages using a similar approach through the `CORE_IMAGE_EXTRA_INSTALL` variable. If you use this variable, only `core-image-*` images are affected.

7.2.2. Customizing Images Using Custom `IMAGE_FEATURES` and `EXTRA_IMAGE_FEATURES`¶

Another method for customizing your image is to enable or disable high-level image features by using the `IMAGE_FEATURES` and `EXTRA_IMAGE_FEATURES` variables. Although the functions for both variables are nearly equivalent, best practices dictate using `IMAGE_FEATURES` from within a recipe and using `EXTRA_IMAGE_FEATURES` from within your `local.conf` file, which is found in the [Build Directory](#).

To understand how these features work, the best reference is `meta/classes/core-image.bbclass`. This class lists out the available `IMAGE_FEATURES` of which most map to package groups while some, such as `debug-tweaks` and `read-only-rootfs`, resolve as general configuration settings.

In summary, the file looks at the contents of the `IMAGE_FEATURES` variable and then maps or configures the feature accordingly. Based on this information, the build system automatically adds the appropriate packages or configurations to the `IMAGE_INSTALL` variable. Effectively, you are enabling extra features by extending the class or creating a custom class for use with specialized image `.bb` files.

Use the `EXTRA_IMAGE_FEATURES` variable from within your local configuration file. Using a separate area from which to enable features with this variable helps you avoid overwriting the features in the image recipe that are enabled with `IMAGE_FEATURES`. The value of `EXTRA_IMAGE_FEATURES` is added to `IMAGE_FEATURES` within `meta/conf/bitbake.conf`.

To illustrate how you can use these variables to modify your image, consider an example that selects the SSH server. The Yocto Project ships with two SSH servers you can use with your images: Dropbear and OpenSSH. Dropbear is a minimal SSH server appropriate for resource-constrained environments, while OpenSSH is a well-known standard SSH server implementation. By default, the `core-image-sato` image is configured to use Dropbear. The `core-image-full-cmdline` and `core-image-lsb` images both include OpenSSH. The `core-image-minimal` image does not contain an SSH server.

You can customize your image and change these defaults. Edit the `IMAGE_FEATURES` variable in your recipe or use the `EXTRA_IMAGE_FEATURES` in your `local.conf` file so that it configures the image you are working with to include `ssh-server-dropbear` or `ssh-server-openssh`.

Note

See the "[Images](#)" section in the Yocto Project Reference Manual for a complete list of image features that ship with the Yocto Project.

7.2.3. Customizing Images Using Custom `.bb` Files¶

You can also customize an image by creating a custom recipe that defines additional software as part of the image. The following example shows the form for the two lines you need:

```
IMAGE_INSTALL = "packagegroup-core-x11-base package1 package2"

inherit core-image
```

Defining the software using a custom recipe gives you total control over the contents of the image. It is important to use the correct names of packages in the `IMAGE_INSTALL` variable. You must use the OpenEmbedded notation and not the Debian notation for the names (e.g. `glibc-dev` instead of `libc6-dev`).

The other method for creating a custom image is to base it on an existing image. For example, if you want to create an image based on `core-image-sato` but add the additional package `strace` to the image, copy the `meta/recipes-sato/images/core-image-sato.bb` to a new `.bb` and add the following line to the end of the copy:

```
IMAGE_INSTALL += "strace"
```

7.2.4. Customizing Images Using Custom Package Groups¶

For complex custom images, the best approach for customizing an image is to create a custom package group recipe that is used to build the image or images. A good example of a package group recipe is `meta/recipes-core/packagegroups/packagegroup-base.bb`.

If you examine that recipe, you see that the `PACKAGES` variable lists the package group packages to produce. The `inherit packagegroup` statement sets appropriate default values and automatically adds `-dev`, `-dbg`, and `-ptest` complementary packages for each package specified in the `PACKAGES` statement.

Note

The `inherit packages` should be located near the top of the recipe, certainly before the `PACKAGES` statement.

For each package you specify in `PACKAGES`, you can use `RDEPENDS` and `RRECOMMENDS` entries to provide a list of packages the parent task package should contain. You can see examples of these further down in the `packagegroup-base.bb` recipe.

Here is a short, fabricated example showing the same basic pieces:

```
DESCRIPTION = "My Custom Package Groups"

inherit packagegroup

PACKAGES = "\
packagegroup-custom-apps \
packagegroup-custom-tools \
"

RDEPENDS_packagegroup-custom-apps = "\
dropbear \
portmap \
psplash"

RDEPENDS_packagegroup-custom-tools = "\
oprofile \
oprofileui-server \
lttng-tools"

RRECOMMENDS_packagegroup-custom-tools = "\
kernel-module-oprofile"
```

In the previous example, two package group packages are created with their dependencies and their recommended package dependencies listed: `packagegroup-custom-apps`, and `packagegroup-custom-tools`. To build an image using these package group packages, you need to add `packagegroup-custom-apps` and/or `packagegroup-custom-tools` to `IMAGE_INSTALL`. For other forms of image dependencies see the other areas of this section.

7.2.5. Customizing an Image Hostname¶

By default, the configured hostname (i.e. `/etc/hostname`) in an image is the same as the machine name. For example, if `MACHINE` equals "qemux86", the configured hostname written to `/etc/hostname` is "qemux86".

You can customize this name by altering the value of the "hostname" variable in the `base-files` recipe using either an append file or a configuration file. Use the following in an append file:

```
hostname="myhostname"
```

Use the following in a configuration file:

```
hostname_pn-base-files = "myhostname"
```

Changing the default value of the variable "hostname" can be useful in certain situations. For example, suppose you need to do extensive testing on an image and you would like to easily identify the image under test from existing images with typical default hostnames. In this situation, you could change the default hostname to "testme", which results in all the images using the name "testme". Once testing is complete and you do not need to rebuild the image for test any longer, you can easily reset the default hostname.

Another point of interest is that if you unset the variable, the image will have no default hostname in the filesystem. Here is an example that unsets the variable in a configuration file:

```
hostname_pn-base-files = ""
```

Having no default hostname in the filesystem is suitable for environments that use dynamic hostnames such as virtual machines.

7.3. Writing a New Recipe¶

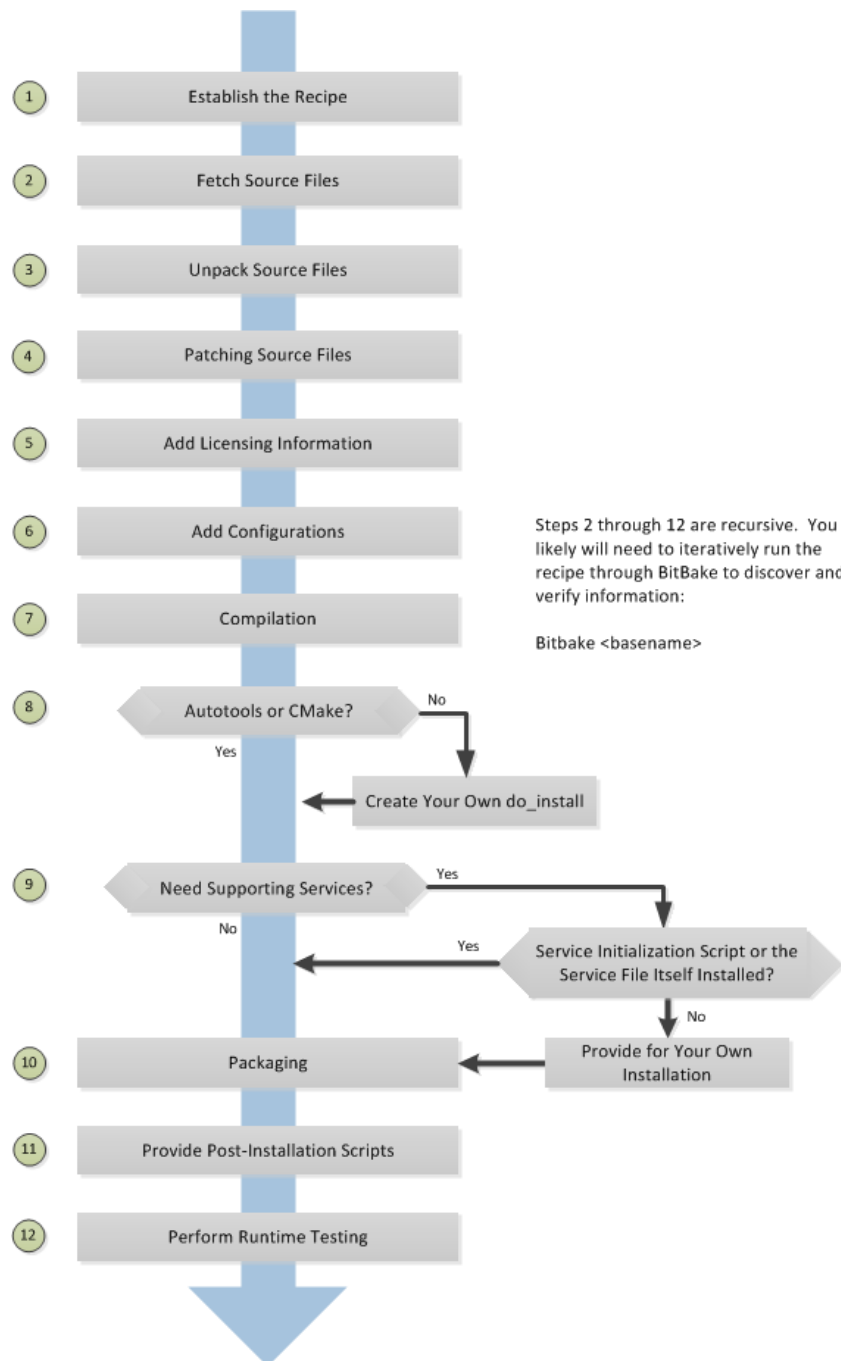
Recipes (`.bb` files) are fundamental components in the Yocto Project environment. Each software component built by the OpenEmbedded build system requires a recipe to define the component. This section describes how to create, write, and test a new recipe.

Note

For information on variables that are useful for recipes and for information about recipe naming issues, see the "[Required](#)" section of the Yocto Project Reference Manual.

7.3.1. Overview¶

The following figure shows the basic process for creating a new recipe. The remainder of the section provides details for the steps.



7.3.2. Locate or Automatically Create a Base Recipe¶

You can always write a recipe from scratch. However, three choices exist that can help you quickly get a start on a new recipe:

- **`devtool add`**: A command that assists in creating a recipe and an environment conducive to development.
- **`recipetool create`**: A command provided by the Yocto Project that automates creation of a base recipe based on the source files.
- **Existing Recipes**: Location and modification of an existing recipe that is similar in function to the recipe you need.

Note

For information on recipe syntax, see the "[Recipe Syntax](#)" section.

7.3.2.1. Creating the Base Recipe Using `devtool add`¶

The `devtool add` command uses the same logic for auto-creating the recipe as `recipetool create`, which is listed below. Additionally, however, `devtool add` sets up an environment that makes it easy for you to patch the source and to make changes to the recipe as is often necessary when adding a recipe to build a new piece of software to be included in a build.

You can find a complete description of the `devtool add` command in the "[A Closer Look at `devtool add`](#)" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

7.3.2.2. Creating the Base Recipe Using `recipetool create`¶

`recipetool create` automates creation of a base recipe given a set of source code files. As long as you can extract or point to the source files, the tool will construct a recipe and automatically configure all pre-build information into the recipe. For example, suppose you have an application that builds using Autotools. Creating the base recipe using `recipetool` results in a recipe that has the pre-build dependencies, license requirements, and checksums configured.

To run the tool, you just need to be in your [Build Directory](#) and have sourced the build environment setup script (i.e. `oe-init-build-env`). To get help on the tool, use the following command:

```
$ recipetool -h
NOTE: Starting bitbake server...
usage: recipetool [-d] [-q] [--color COLOR] [-h] <subcommand> ...

OpenEmbedded recipe tool

options:
  -d, --debug          Enable debug output
  -q, --quiet          Print only errors
  --color COLOR        Colorize output (where COLOR is auto, always, never)
  -h, --help           show this help message and exit

subcommands:
  create              Create a new recipe
  newappend           Create a bbappend for the specified target in the specified
                     layer
  setvar              Set a variable within a recipe
  appendfile          Create/update a bbappend to replace a target file
  appendsrcfiles      Create/update a bbappend to add or replace source files
  appendsrcfile       Create/update a bbappend to add or replace a source file
  Use recipetool <subcommand> --help to get help on a specific command
```

Running `recipetool create -o OUTFILE` creates the base recipe and locates it properly in the layer that contains your source files. Following are some syntax examples:

Use this syntax to generate a recipe based on *source*. Once generated, the recipe resides in the existing source code layer:

```
recipetool create -o OUTFILE source
```

Use this syntax to generate a recipe using code that you extract from *source*. The extracted code is placed in its own layer defined by *EXTERNALSRC*.

```
recipetool create -o OUTFILE -x EXTERNALSRC source
```

Use this syntax to generate a recipe based on *source*. The options direct `recipetool` to generate debugging information. Once generated, the recipe resides in the existing source code layer:

```
recipetool create -d -o OUTFILE source
```

7.3.2.3. Locating and Using a Similar Recipe¶

Before writing a recipe from scratch, it is often useful to discover whether someone else has already written one that meets (or comes close to meeting) your needs. The Yocto Project and OpenEmbedded communities maintain many recipes that might be candidates for what you are doing. You can find a good central index of these recipes in the [OpenEmbedded Layer Index](#).

Working from an existing recipe or a skeleton recipe is the best way to get started. Here are some points on both methods:

- **Locate and modify a recipe that is close to what you want to do:** This method works when you are familiar with the current recipe space. The method does not work so well for those new to the Yocto Project or writing recipes.

Some risks associated with this method are using a recipe that has areas totally unrelated to what you are trying to accomplish with your recipe, not recognizing areas of the recipe that you might have to add from scratch, and so forth. All these risks stem from unfamiliarity with the existing recipe space.

- **Use and modify the following skeleton recipe:** If for some reason you do not want to use `recipetool` and you cannot find an existing recipe that is close to meeting your needs, you can use the following structure to provide the fundamental areas of a new recipe.

```
DESCRIPTION = ""
HOMEPAGE = ""
LICENSE = ""
SECTION = ""
DEPENDS = ""
LIC_FILES_CHKSUM = ""

SRC_URI = ""
```

7.3.3. Storing and Naming the Recipe¶

Once you have your base recipe, you should put it in your own layer and name it appropriately. Locating it correctly ensures that the OpenEmbedded build system can find it when you use BitBake to process the recipe.

- **Storing Your Recipe:** The OpenEmbedded build system locates your recipe through the layer's `conf/layer.conf` file and the `BBFILES` variable. This variable sets up a path from which the build system can locate recipes. Here is the typical use:

```
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
           ${LAYERDIR}/recipes-*/*/*.bbappend"
```

Consequently, you need to be sure you locate your new recipe inside your layer such that it can be found.

You can find more information on how layers are structured in the ["Understanding and Creating Layers"](#) section.

- **Naming Your Recipe:** When you name your recipe, you need to follow this naming convention:

```
basename_version.bb
```

Use lower-cased characters and do not include the reserved suffixes `-native`, `-cross`, `-initial`, or `-dev` casually (i.e. do not use them as part of your recipe name unless the string applies). Here are some examples:

```
cups_1.7.0.bb
gawk_4.0.2.bb
irssi_0.8.16-rc1.bb
```

7.3.4. Running a Build on the Recipe ¶

Creating a new recipe is usually an iterative process that requires using BitBake to process the recipe multiple times in order to progressively discover and add information to the recipe file.

Assuming you have sourced the build environment setup script (i.e. `oe-init-build-env`) and you are in the [Build Directory](#), use BitBake to process your recipe. All you need to provide is the *basename* of the recipe as described in the previous section:

```
$ bitbake basename
```

During the build, the OpenEmbedded build system creates a temporary work directory for each recipe (`${WORKDIR}`) where it keeps extracted source files, log files, intermediate compilation and packaging files, and so forth.

The path to the per-recipe temporary work directory depends on the context in which it is being built. The quickest way to find this path is to have BitBake return it by running the following:

```
$ bitbake -e basename | grep ^WORKDIR=
```

As an example, assume a Source Directory top-level folder named `poky`, a default Build Directory at `poky/build`, and a `qemux86-poky-linux` machine target system. Furthermore, suppose your recipe is named `foo_1.3.0.bb`. In this case, the work directory the build system uses to build the package would be as follows:

```
poky/build/tmp/work/qemux86-poky-linux/foo/1.3.0-r0
```

Inside this directory you can find sub-directories such as `image`, `packages-split`, and `temp`. After the build, you can examine these to determine how well the build went.

Note

You can find log files for each task in the recipe's `temp` directory (e.g. `poky/build/tmp/work/qemux86-poky-linux/foo/1.3.0-r0/temp`). Log files are named `log.taskname` (e.g. `log.do_configure`, `log.do_fetch`, and `log.do_compile`).

You can find more information about the build process in "[The Yocto Project Development Environment](#)" chapter of the Yocto Project Overview and Concepts Manual.

7.3.5. Fetching Code ¶

The first thing your recipe must do is specify how to fetch the source files. Fetching is controlled mainly through the `SRC_URI` variable. Your recipe must have a `SRC_URI` variable that points to where the source is located. For a graphical representation of source locations, see the "[Sources](#)" section in the Yocto Project Overview and Concepts Manual.

The `do_fetch` task uses the prefix of each entry in the `SRC_URI` variable value to determine which `fetcher` to use to get your source files. It is the `SRC_URI` variable that triggers the fetcher. The `do_patch` task uses the variable after source is fetched to apply patches. The OpenEmbedded build system uses [FILESOVERRIDES](#) for scanning directory locations for local files in `SRC_URI`.

The `SRC_URI` variable in your recipe must define each unique location for your source files. It is good practice to not hard-code pathnames in an URL used in `SRC_URI`. Rather than hard-code these paths, use `${PV}`, which causes the fetch process to use the version specified in the recipe filename. Specifying the version in this manner means that upgrading the recipe to a future version is as simple as renaming the recipe to match the new version.

Here is a simple example from the `meta/recipes-devtools/cdrtools/cdrtools-native_3.01a20.bb` recipe where the source comes from a single tarball. Notice the use of the `PV` variable:

```
SRC_URI = "ftp://ftp.berlios.de/pub/cdrecord/alpha/cdrtools-${PV}.tar.bz2"
```

Files mentioned in `SRC_URI` whose names end in a typical archive extension (e.g. `.tar`, `.tar.gz`, `.tar.bz2`, `.zip`, and so forth), are automatically extracted during the `do_unpack` task. For another example that specifies these types of files, see the "[Autotooled Package](#)" section.

Another way of specifying source is from an SCM. For Git repositories, you must specify `SRCREV` and you should specify `PV` to include the revision with `SRCPV`. Here is an example from the recipe `meta/recipes-kernel/blktrace/blktrace_git.bb`:

```
SRCREV = "d6918c8832793b4205ed3bfede78c2f915c23385"

PR = "r6"
PV = "1.0.5+git${SRCPV}"

SRC_URI = "git://git.kernel.dk/blktrace.git \
           file://ldflags.patch"
```

If your `SRC_URI` statement includes URLs pointing to individual files fetched from a remote server other than a version control system, BitBake attempts to verify the files against checksums defined in your recipe to ensure they have not been tampered with or otherwise modified since the recipe was written. Two checksums are used: `SRC_URI [md5sum]` and `SRC_URI [sha256sum]`.

If your `SRC_URI` variable points to more than a single URL (excluding SCM URLs), you need to provide the `md5` and `sha256` checksums for each URL. For these cases, you provide a name for each URL as part of the `SRC_URI` and then reference that name in the subsequent checksum statements. Here is an example:

```
SRC_URI = "${DEBIAN_MIRROR}/main/a/apmd/apmd_3.2.2.orig.tar.gz;name=tarball \
${DEBIAN_MIRROR}/main/a/apmd/apmd_${PV}.diff.gz;name=patch"

SRC_URI[tarball.md5sum] = "b1e6309e8331e0f4e6efd311c2d97fa8"
SRC_URI[tarball.sha256sum] = "7f7d9f60b7766b852881d40b8ff91d8e39fccb0d1d913102a5c75a2dbb52332d"

SRC_URI[patch.md5sum] = "57e1b689264ea80f78353519eece0c92"
SRC_URI[patch.sha256sum] = "7905ff96be93d725544d0040e425c42f9c05580db3c272f11cff75b9aa89d430"
```

Proper values for `md5` and `sha256` checksums might be available with other signatures on the download page for the upstream source (e.g. `md5`, `sha1`, `sha256`, GPG, and so forth). Because the OpenEmbedded build system only deals with `sha256sum` and `md5sum`, you should verify all the signatures you find by hand.

If no `SRC_URI` checksums are specified when you attempt to build the recipe, or you provide an incorrect checksum, the build will produce an error for each missing or incorrect checksum. As part of the error message, the build system provides the checksum string corresponding to the fetched file. Once you have the correct checksums, you can copy and paste them into your recipe and then run the build again to continue.

Note

As mentioned, if the upstream source provides signatures for verifying the downloaded source code, you should verify those manually before setting the checksum values in the recipe and continuing with the build.

This final example is a bit more complicated and is from the `meta/recipes-sato/rxvt-unicode/rxvt-unicode_9.20.bb` recipe. The example's `SRC_URI` statement identifies multiple files as the source files for the recipe: a tarball, a patch file, a desktop file, and an icon.

```
SRC_URI = "http://dist.schmorp.de/rxvt-unicode/Attic/rxvt-unicode-${PV}.tar.bz2 \
file://xwc.patch \
file://rxvt.desktop \
file://rxvt.png"
```

When you specify local files using the `file://` URI protocol, the build system fetches files from the local machine. The path is relative to the `FILESPATH` variable and searches specific directories in a certain order: `${BP}`, `${BPN}`, and `files`. The directories are assumed to be subdirectories of the directory in which the recipe or append file resides. For another example that specifies these types of files, see the "[Single .c File Package \(Hello World!\)](#)" section.

The previous example also specifies a patch file. Patch files are files whose names usually end in `.patch` or `.diff` but can end with compressed suffixes such as `diff.gz` and `patch.bz2`, for example. The build system automatically applies patches as described in the "[Patching Code](#)" section.

7.3.6. Unpacking Code¶

During the build, the `do_unpack` task unpacks the source with `${S}` pointing to where it is unpacked.

If you are fetching your source files from an upstream source archived tarball and the tarball's internal structure matches the common convention of a top-level subdirectory named `${BPN}-${PV}`, then you do not need to set `S`. However, if `SRC_URI` specifies to fetch source from an archive that does not use this convention, or from an SCM like Git or Subversion, your recipe needs to define `S`.

If processing your recipe using BitBake successfully unpacks the source files, you need to be sure that the directory pointed to by `${S}` matches the structure of the source.

7.3.7. Patching Code¶

Sometimes it is necessary to patch code after it has been fetched. Any files mentioned in `SRC_URI` whose names end in `.patch` or `.diff` or compressed versions of these suffixes (e.g. `diff.gz` are treated as patches. The `do_patch` task automatically applies these patches.

The build system should be able to apply patches with the `-p1` option (i.e. one directory level in the path will be stripped off). If your patch needs to have more directory levels stripped off, specify the number of levels using the `"striplevel"` option in the `SRC_URI` entry for the patch. Alternatively, if your patch needs to be applied in a specific subdirectory that is not specified in the patch file, use the `"patchdir"` option in the entry.

As with all local files referenced in `SRC_URI` using `file://`, you should place patch files in a directory next to the recipe either named the same as the base name of the recipe (`BP` and `BPN`) or `"files"`.

7.3.8. Licensing¶

Your recipe needs to have both the `LICENSE` and `LIC_FILES_CHKSUM` variables:

- **LICENSE:** This variable specifies the license for the software. If you do not know the license under which the software you are building is distributed, you should go to the source code and look for that information. Typical files containing this information include `COPYING`, `LICENSE`, and `README` files. You could also find the information near the top of a source file. For example, given a piece of software licensed under the GNU General Public License version 2, you would set `LICENSE` as follows:

```
LICENSE = "GPLv2"
```

The licenses you specify within `LICENSE` can have any name as long as you do not use spaces, since spaces are used as separators between license names. For standard licenses, use the names of the files in `meta/files/common-licenses/` or the `SPDXLICENSEMAP` flag names defined in `meta/conf/licenses.conf`.

- **`LIC_FILES_CHKSUM`**: The OpenEmbedded build system uses this variable to make sure the license text has not changed. If it has, the build produces an error and it affords you the chance to figure it out and correct the problem.

You need to specify all applicable licensing files for the software. At the end of the configuration step, the build process will compare the checksums of the files to be sure the text has not changed. Any differences result in an error with the message containing the current checksum. For more explanation and examples of how to set the `LIC_FILES_CHKSUM` variable, see the "Tracking License Changes" section.

To determine the correct checksum string, you can list the appropriate files in the `LIC_FILES_CHKSUM` variable with incorrect md5 strings, attempt to build the software, and then note the resulting error messages that will report the correct md5 strings. See the "[Fetching Code](#)" section for additional information.

Here is an example that assumes the software has a `COPYING` file:

```
LIC_FILES_CHKSUM = "file://COPYING;md5=xxx"
```

When you try to build the software, the build system will produce an error and give you the correct string that you can substitute into the recipe file for a subsequent build.

7.3.9. Dependencies¶

Most software packages have a short list of other packages that they require, which are called dependencies. These dependencies fall into two main categories: build-time dependencies, which are required when the software is built; and runtime dependencies, which are required to be installed on the target in order for the software to run.

Within a recipe, you specify build-time dependencies using the `DEPENDS` variable. Although nuances exist, items specified in `DEPENDS` should be names of other recipes. It is important that you specify all build-time dependencies explicitly. If you do not, due to the parallel nature of BitBake's execution, you can end up with a race condition where the dependency is present for one task of a recipe (e.g. `do_configure`) and then gone when the next task runs (e.g. `do_compile`).

Another consideration is that configure scripts might automatically check for optional dependencies and enable corresponding functionality if those dependencies are found. This behavior means that to ensure deterministic results and thus avoid more race conditions, you need to either explicitly specify these dependencies as well, or tell the configure script explicitly to disable the functionality. If you wish to make a recipe that is more generally useful (e.g. publish the recipe in a layer for others to use), instead of hard-disabling the functionality, you can use the `PACKAGECONFIG` variable to allow functionality and the corresponding dependencies to be enabled and disabled easily by other users of the recipe.

Similar to build-time dependencies, you specify runtime dependencies through a variable - `RDEPENDS`, which is package-specific. All variables that are package-specific need to have the name of the package added to the end as an override. Since the main package for a recipe has the same name as the recipe, and the recipe's name can be found through the `${PN}` variable, then you specify the dependencies for the main package by setting `RDEPENDS_${PN}`. If the package were named `${PN}-tools`, then you would set `RDEPENDS_${PN}-tools`, and so forth.

Some runtime dependencies will be set automatically at packaging time. These dependencies include any shared library dependencies (i.e. if a package "example" contains "libexample" and another package "mypackage" contains a binary that links to "libexample" then the OpenEmbedded build system will automatically add a runtime dependency to "mypackage" on "example"). See the "[Automatically Added Runtime Dependencies](#)" section in the Yocto Project Overview and Concepts Manual for further details.

7.3.10. Configuring the Recipe¶

Most software provides some means of setting build-time configuration options before compilation. Typically, setting these options is accomplished by running a configure script with options, or by modifying a build configuration file.

Note

As of Yocto Project Release 1.7, some of the core recipes that package binary configuration scripts now disable the scripts due to the scripts previously requiring error-prone path substitution. The OpenEmbedded build system uses `pkg-config` now, which is much more robust. You can find a list of the `*-config` scripts that are disabled in the "[Binary Configuration Scripts Disabled](#)" section in the Yocto Project Reference Manual.

A major part of build-time configuration is about checking for build-time dependencies and possibly enabling optional functionality as a result. You need to specify any build-time dependencies for the software you are building in your recipe's `DEPENDS` value, in terms of other recipes that satisfy those dependencies. You can often find build-time or runtime dependencies described in the software's documentation.

The following list provides configuration items of note based on how your software is built:

- **`Autotools`**: If your source files have a `configure.ac` file, then your software is built using Autotools. If this is the case, you just need to worry about modifying the configuration.

When using Autotools, your recipe needs to inherit the `autotools` class and your recipe does not have to contain a `do_configure` task. However, you might still want to make some adjustments. For example, you can set `EXTRA_OECONF` or `PACKAGECONFIG_CONFARGS` to pass any needed configure options that are specific to the recipe.

- **`CMake`**: If your source files have a `CMakeLists.txt` file, then your software is built using CMake. If this is the case, you just need to worry about modifying the configuration.

When you use CMake, your recipe needs to inherit the `cmake` class and your recipe does not have to contain a `do_configure` task. You can make some adjustments by setting `EXTRA_OECMAKE` to pass any needed configure options that are specific to the recipe.

Note

If you need to install one or more custom CMake toolchain files that are supplied by the application you are building, install the files to `${D}${datadir}/cmake/Modules` during `do_install`.

- **Other:** If your source files do not have a `configure.ac` or `CMakeLists.txt` file, then your software is built using some method other than Autotools or CMake. If this is the case, you normally need to provide a `do_configure` task in your recipe unless, of course, there is nothing to configure.

Even if your software is not being built by Autotools or CMake, you still might not need to deal with any configuration issues. You need to determine if configuration is even a required step. You might need to modify a Makefile or some configuration file used for the build to specify necessary build options. Or, perhaps you might need to run a provided, custom configure script with the appropriate options.

For the case involving a custom configure script, you would run `./configure --help` and look for the options you need to set.

Once configuration succeeds, it is always good practice to look at the `log.do_configure` file to ensure that the appropriate options have been enabled and no additional build-time dependencies need to be added to `DEPENDS`. For example, if the configure script reports that it found something not mentioned in `DEPENDS`, or that it did not find something that it needed for some desired optional functionality, then you would need to add those to `DEPENDS`. Looking at the log might also reveal items being checked for, enabled, or both that you do not want, or items not being found that are in `DEPENDS`, in which case you would need to look at passing extra options to the configure script as needed. For reference information on configure options specific to the software you are building, you can consult the output of the `./configure --help` command within `${S}` or consult the software's upstream documentation.

7.3.11. Using Headers to Interface with Devices¶

If your recipe builds an application that needs to communicate with some device or needs an API into a custom kernel, you will need to provide appropriate header files. Under no circumstances should you ever modify the existing `meta/recipes-kernel/linux-libc-headers/linux-libc-headers.inc` file. These headers are used to build `libc` and must not be compromised with custom or machine-specific header information. If you customize `libc` through modified headers all other applications that use `libc` thus become affected.

Warning

Never copy and customize the `libc` header file (i.e. `meta/recipes-kernel/linux-libc-headers/linux-libc-headers.inc`).

The correct way to interface to a device or custom kernel is to use a separate package that provides the additional headers for the driver or other unique interfaces. When doing so, your application also becomes responsible for creating a dependency on that specific provider.

Consider the following:

- Never modify `linux-libc-headers.inc`. Consider that file to be part of the `libc` system, and not something you use to access the kernel directly. You should access `libc` through specific `libc` calls.
- Applications that must talk directly to devices should either provide necessary headers themselves, or establish a dependency on a special headers package that is specific to that driver.

For example, suppose you want to modify an existing header that adds I/O control or network support. If the modifications are used by a small number programs, providing a unique version of a header is easy and has little impact. When doing so, bear in mind the guidelines in the previous list.

Note

If for some reason your changes need to modify the behavior of the `libc`, and subsequently all other applications on the system, use a `.bbappend` to modify the `linux-kernel-headers.inc` file. However, take care to not make the changes machine specific.

Consider a case where your kernel is older and you need an older `libc` ABI. The headers installed by your recipe should still be a standard mainline kernel, not your own custom one.

When you use custom kernel headers you need to get them from `STAGING_KERNEL_DIR`, which is the directory with kernel headers that are required to build out-of-tree modules. Your recipe will also need the following:

```
do_configure[depends] += "virtual/kernel:do_shared_workdir"
```

7.3.12. Compilation¶

During a build, the `do_compile` task happens after source is fetched, unpacked, and configured. If the recipe passes through `do_compile` successfully, nothing needs to be done.

However, if the compile step fails, you need to diagnose the failure. Here are some common issues that cause failures.

Note

For cases where improper paths are detected for configuration files or for when libraries/headers cannot be found, be sure you are using the more robust `pkg-config`. See the note in section "[Configuring the Recipe](#)" for additional information.

- **Parallel build failures:** These failures manifest themselves as intermittent errors, or errors reporting that a file or directory that should be created by some other part of the build process could not be found. This type of failure can occur even if, upon inspection, the file or directory does exist after the build has failed, because that part of the build process happened in the wrong order.

To fix the problem, you need to either satisfy the missing dependency in the Makefile or whatever script produced the Makefile, or (as a workaround) set `PARALLEL_MAKE` to an empty string:

```
PARALLEL_MAKE = ""
```

For information on parallel Makefile issues, see the "[Debugging Parallel Make Races](#)" section.

- **Improper host path usage:** This failure applies to recipes building for the target or `nativesdk` only. The failure occurs when the compilation process uses improper headers, libraries, or other files from the host system when cross-compiling for the target.

To fix the problem, examine the `log.do_compile` file to identify the host paths being used (e.g. `/usr/include`, `/usr/lib`, and so forth) and then either add configure options, apply a patch, or do both.

- **Failure to find required libraries/headers:** If a build-time dependency is missing because it has not been declared in `DEPENDS`, or because the dependency exists but the path used by the build process to find the file is incorrect and the configure step did not detect it, the compilation process could fail. For either of these failures, the compilation process notes that files could not be found. In these cases, you need to go back and add additional options to the configure script as well as possibly add additional build-time dependencies to `DEPENDS`.

Occasionally, it is necessary to apply a patch to the source to ensure the correct paths are used. If you need to specify paths to find files staged into the sysroot from other recipes, use the variables that the OpenEmbedded build system provides (e.g. `STAGING_BINDIR`, `STAGING_INCDIR`, `STAGING_DATADIR`, and so forth).

7.3.13. Installing¶

During `do_install`, the task copies the built files along with their hierarchy to locations that would mirror their locations on the target device. The installation process copies files from the `${S}`, `${B}`, and `${WORKDIR}` directories to the `${D}` directory to create the structure as it should appear on the target system.

How your software is built affects what you must do to be sure your software is installed correctly. The following list describes what you must do for installation depending on the type of build system used by the software being built:

- **Autotools and CMake:** If the software your recipe is building uses Autotools or CMake, the OpenEmbedded build system understands how to install the software. Consequently, you do not have to have a `do_install` task as part of your recipe. You just need to make sure the install portion of the build completes with no issues. However, if you wish to install additional files not already being installed by `make install`, you should do this using a `do_install_append` function using the `install` command as described in the "Manual" bulleted item later in this list.
- **Other (using `make install`):** You need to define a `do_install` function in your recipe. The function should call `oe_runmake install` and will likely need to pass in the destination directory as well. How you pass that path is dependent on how the Makefile being run is written (e.g. `DESTDIR=${D}`, `PREFIX=${D}`, `INSTALLROOT=${D}`, and so forth).

For an example recipe using `make install`, see the "[Makefile-Based Package](#)" section.

- **Manual:** You need to define a `do_install` function in your recipe. The function must first use `install -d` to create the directories under `${D}`. Once the directories exist, your function can use `install` to manually install the built software into the directories.

You can find more information on `install` at http://www.gnu.org/software/coreutils/manual/html_node/install-invocation.html.

For the scenarios that do not use Autotools or CMake, you need to track the installation and diagnose and fix any issues until everything installs correctly. You need to look in the default location of `${D}`, which is `${WORKDIR}/image`, to be sure your files have been installed correctly.

Notes

- During the installation process, you might need to modify some of the installed files to suit the target layout. For example, you might need to replace hard-coded paths in an `initscript` with values of variables provided by the build system, such as replacing `/usr/bin/` with `${bindir}`. If you do perform such modifications during `do_install`, be sure to modify the destination file after copying rather than before copying. Modifying after copying ensures that the build system can re-execute `do_install` if needed.
- `oe_runmake install`, which can be run directly or can be run indirectly by the `autotools` and `cmake` classes, runs `make install` in parallel. Sometimes, a Makefile can have missing dependencies between targets that can result in race conditions. If you experience intermittent failures during `do_install`, you might be able to work around them by disabling parallel Makefile installs by adding the following to the recipe:

```
PARALLEL_MAKEINST = ""
```

See `PARALLEL_MAKEINST` for additional information.

- If you need to install one or more custom CMake toolchain files that are supplied by the application you are building, install the files to `${D}${datadir}/cmake/Modules` during `do_install`.

7.3.14. Enabling System Services¶

If you want to install a service, which is a process that usually starts on boot and runs in the background, then you must include some additional definitions in your recipe.

If you are adding services and the service initialization script or the service file itself is not installed, you must provide for that installation in your recipe using a `do_install_append` function. If your recipe already has a `do_install` function, update the function near its end rather than adding an additional `do_install_append` function.

When you create the installation for your services, you need to accomplish what is normally done by `make install`. In other words, make sure your installation arranges the output similar to how it is arranged on the target system.

The OpenEmbedded build system provides support for starting services two different ways:

- **SysVinit:** SysVinit is a system and service manager that manages the init system used to control the very basic functions of your system. The init program is the first program started by the Linux kernel when the system boots. Init then controls the startup, running and shutdown of all other programs.

To enable a service using SysVinit, your recipe needs to inherit the `update-rc.d` class. The class helps facilitate safely installing the package on the target.

You will need to set the `INITSCRIPT_PACKAGES`, `INITSCRIPT_NAME`, and `INITSCRIPT_PARAMS` variables within your recipe.

- **systemd:** System Management Daemon (systemd) was designed to replace SysVinit and to provide enhanced management of services. For more information on systemd, see the systemd homepage at <http://freedesktop.org/wiki/Software/systemd/>.

To enable a service using systemd, your recipe needs to inherit the `systemd` class. See the `systemd.bbclass` file located in your [Source Directory](#) section for more information.

7.3.15. Packaging¶

Successful packaging is a combination of automated processes performed by the OpenEmbedded build system and some specific steps you need to take. The following list describes the process:

- **Splitting Files:** The `do_package` task splits the files produced by the recipe into logical components. Even software that produces a single binary might still have debug symbols, documentation, and other logical components that should be split out. The `do_package` task ensures that files are split up and packaged correctly.
- **Running QA Checks:** The `insane` class adds a step to the package generation process so that output quality assurance checks are generated by the OpenEmbedded build system. This step performs a range of checks to be sure the build's output is free of common problems that show up during runtime. For information on these checks, see the `insane` class and the "[QA Error and Warning Messages](#)" chapter in the Yocto Project Reference Manual.
- **Hand-Checking Your Packages:** After you build your software, you need to be sure your packages are correct. Examine the `${WORKDIR}/packages-split` directory and make sure files are where you expect them to be. If you discover problems, you can set `PACKAGES`, `FILES`, `do_install(_append)`, and so forth as needed.
- **Splitting an Application into Multiple Packages:** If you need to split an application into several packages, see the "[Splitting an Application into Multiple Packages](#)" section for an example.
- **Installing a Post-Installation Script:** For an example showing how to install a post-installation script, see the "[Post-Installation Scripts](#)" section.
- **Marking Package Architecture:** Depending on what your recipe is building and how it is configured, it might be important to mark the packages produced as being specific to a particular machine, or to mark them as not being specific to a particular machine or architecture at all.

By default, packages apply to any machine with the same architecture as the target machine. When a recipe produces packages that are machine-specific (e.g. the `MACHINE` value is passed into the configure script or a patch is applied only for a particular machine), you should mark them as such by adding the following to the recipe:

```
PACKAGE_ARCH = "${MACHINE_ARCH}"
```

On the other hand, if the recipe produces packages that do not contain anything specific to the target machine or architecture at all (e.g. recipes that simply package script files or configuration files), you should use the `allarch` class to do this for you by adding this to your recipe:

```
inherit allarch
```

Ensuring that the package architecture is correct is not critical while you are doing the first few builds of your recipe. However, it is important in order to ensure that your recipe rebuilds (or does not rebuild) appropriately in response to changes in configuration, and to ensure that you get the appropriate packages installed on the target machine, particularly if you run separate builds for more than one target machine.

7.3.16. Sharing Files Between Recipes¶

Recipes often need to use files provided by other recipes on the build host. For example, an application linking to a common library needs access to the library itself and its associated headers. The way this access is accomplished is by populating a sysroot with files. Each recipe has two sysroots in its work directory, one for target files (`recipe-sysroot`) and one for files that are native to the build host (`recipe-sysroot-native`).

Note

You could find the term "staging" used within the Yocto project regarding files populating sysroots (e.g. the [STAGING_DIR](#) variable).

Recipes should never populate the sysroot directly (i.e. write files into sysroot). Instead, files should be installed into standard locations during the [do_install](#) task within the `${D}` directory. The reason for this limitation is that almost all files that populate the sysroot are cataloged in manifests in order to ensure the files can be removed later when a recipe is either modified or removed. Thus, the sysroot is able to remain free from stale files.

A subset of the files installed by the [do_install](#) task are used by the [do_populate_sysroot](#) task as defined by the the [SYSROOT_DIRS](#) variable to automatically populate the sysroot. It is possible to modify the list of directories that populate the sysroot. The following example shows how you could add the `/opt` directory to the list of directories within a recipe:

```
SYSROOT_DIRS += "/opt"
```

For a more complete description of the [do_populate_sysroot](#) task and its associated functions, see the [staging](#) class.

7.3.17. Using Virtual Providers¶

Prior to a build, if you know that several different recipes provide the same functionality, you can use a virtual provider (i.e. `virtual/*`) as a placeholder for the actual provider. The actual provider is determined at build-time.

A common scenario where a virtual provider is used would be for the kernel recipe. Suppose you have three kernel recipes whose [PN](#) values map to `kernel-big`, `kernel-mid`, and `kernel-small`. Furthermore, each of these recipes in some way uses a [PROVIDES](#) statement that essentially identifies itself as being able to provide `virtual/kernel`. Here is one way through the [kernel](#) class:

```
PROVIDES += "${@ "virtual/kernel" if (d.getVar("KERNEL_PACKAGE_NAME") == "kernel") else "" }"
```

Any recipe that inherits the `kernel` class is going to utilize a [PROVIDES](#) statement that identifies that recipe as being able to provide the `virtual/kernel` item.

Now comes the time to actually build an image and you need a kernel recipe, but which one? You can configure your build to call out the kernel recipe you want by using the [PREFERRED_PROVIDER](#) variable. As an example, consider the `x86-base.inc` include file, which is a machine (i.e. [MACHINE](#)) configuration file. This include file is the reason all x86-based machines use the `linux-yocto` kernel. Here are the relevant lines from the include file:

```
PREFERRED_PROVIDER_virtual/kernel ??= "linux-yocto"
PREFERRED_VERSION_linux-yocto ??= "4.15%"
```

When you use a virtual provider, you do not have to "hard code" a recipe name as a build dependency. You can use the [DEPENDS](#) variable to state the build is dependent on `virtual/kernel` for example:

```
DEPENDS = "virtual/kernel"
```

During the build, the OpenEmbedded build system picks the correct recipe needed for the `virtual/kernel` dependency based on the [PREFERRED_PROVIDER](#) variable. If you want to use the small kernel mentioned at the beginning of this section, configure your build as follows:

```
PREFERRED_PROVIDER_virtual/kernel ??= "kernel-small"
```

Note

Any recipe that [PROVIDES](#) a `virtual/*` item that is ultimately not selected through [PREFERRED_PROVIDER](#) does not get built. Preventing these recipes from building is usually the desired behavior since this mechanism's purpose is to select between mutually exclusive alternative providers.

The following lists specific examples of virtual providers:

- `virtual/kernel`: Provides the name of the kernel recipe to use when building a kernel image.
- `virtual/bootloader`: Provides the name of the bootloader to use when building an image.
- `virtual/mesa`: Provides `gbm.pc`.
- `virtual/egl`: Provides `egl.pc` and possibly `wayland-egl.pc`.
- `virtual/libgl`: Provides `gl.pc` (i.e. `libGL`).
- `virtual/libgles1`: Provides `glesv1_cm.pc` (i.e. `libGLESv1_CM`).
- `virtual/libgles2`: Provides `glesv2.pc` (i.e. `libGLESv2`).

7.3.18. Properly Versioning Pre-Release Recipes¶

Sometimes the name of a recipe can lead to versioning problems when the recipe is upgraded to a final release. For example, consider the `irssi_0.8.16-rc1.bb` recipe file in the list of example recipes in the ["Storing and Naming the Recipe"](#) section. This recipe is at a release candidate stage (i.e. "rc1"). When the recipe is released, the recipe filename becomes `irssi_0.8.16.bb`. The version change from `0.8.16-rc1` to `0.8.16` is seen as a decrease by the build system and package managers, so the resulting packages will not correctly trigger an upgrade.

In order to ensure the versions compare properly, the recommended convention is to set `PV` within the recipe to `"previous_version+current_version"`. You can use an additional variable so that you can use the current version elsewhere. Here is an example:

```
REALPV = "0.8.16-rc1"
PV = "0.8.15+${REALPV}"
```

7.3.19. Post-Installation Scripts¶

Post-installation scripts run immediately after installing a package on the target or during image creation when a package is included in an image. To add a post-installation script to a package, add a `pkg_postinst_PACKAGENAME()` function to the recipe file (`.bb`) and replace `PACKAGENAME` with the name of the package you want to attach to the `postinst` script. To apply the post-installation script to the main package for the recipe, which is usually what is required, specify `${PN}` in place of `PACKAGENAME`.

A post-installation function has the following structure:

```
pkg_postinst_PACKAGENAME() {
# Commands to carry out
}
```

The script defined in the post-installation function is called when the root filesystem is created. If the script succeeds, the package is marked as installed. If the script fails, the package is marked as unpacked and the script is executed when the image boots again.

Note

Any RPM post-installation script that runs on the target should return a 0 exit code. RPM does not allow non-zero exit codes for these scripts, and the RPM package manager will cause the package to fail installation on the target.

Sometimes it is necessary for the execution of a post-installation script to be delayed until the first boot. For example, the script might need to be executed on the device itself. To delay script execution until boot time, you must explicitly mark post installs to defer to the target. You can use `pkg_postinst_ontarget()` or call `postinst-intercepts defer_to_first_boot` from `pkg_postinst()`. Any failure of a `pkg_postinst()` script (including exit 1) triggers an error during the `do_rootfs` task.

If you have recipes that use `pkg_postinst` function and they require the use of non-standard native tools that have dependencies during rootfs construction, you need to use the `PACKAGE_WRITE_DEPS` variable in your recipe to list these tools. If you do not use this variable, the tools might be missing and execution of the post-installation script is deferred until first boot. Deferring the script to first boot is undesirable and for read-only rootfs impossible.

Note

Equivalent support for pre-install, pre-uninstall, and post-uninstall scripts exist by way of `pkg_preinst`, `pkg_prerm`, and `pkg_postrm`, respectively. These scripts work in exactly the same way as does `pkg_postinst` with the exception that they run at different times. Also, because of when they run, they are not applicable to being run at image creation time like `pkg_postinst`.

7.3.20. Testing¶

The final step for completing your recipe is to be sure that the software you built runs correctly. To accomplish runtime testing, add the build's output packages to your image and test them on the target.

For information on how to customize your image by adding specific packages, see the ["Customizing Images"](#) section.

7.3.21. Examples¶

To help summarize how to write a recipe, this section provides some examples given various scenarios:

- Recipes that use local files
- Using an Autotooled package
- Using a Makefile-based package
- Splitting an application into multiple packages
- Adding binaries to an image

7.3.21.1. Single .c File Package (Hello World!)¶

Building an application from a single file that is stored locally (e.g. under `files`) requires a recipe that has the file listed in the `SRC_URI` variable. Additionally, you need to manually write the `do_compile` and `do_install` tasks. The `S` variable defines the directory containing the source code, which is set to `WORKDIR` in this case - the directory BitBake uses for the build.

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

SRC_URI = "file://helloworld.c"

S = "${WORKDIR}"
```

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

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

By default, the `helloworld`, `helloworld-dbg`, and `helloworld-dev` packages are built. For information on how to customize the packaging process, see the "[Splitting an Application into Multiple Packages](#)" section.

7.3.21.2. Autotooled Package

Applications that use Autotools such as `autoconf` and `automake` require a recipe that has a source archive listed in [SRC_URI](#) and also inherit the `autotools` class, which contains the definitions of all the steps needed to build an Autotool-based application. The result of the build is automatically packaged. And, if the application uses NLS for localization, packages with local information are generated (one package per language). Following is one example: (`hello_2.3.bb`)

```
SUMMARY = "GNU Helloworld application"
SECTION = "examples"
LICENSE = "GPLv2+"
LIC_FILES_CHKSUM = "file://COPYING;md5=751419260aa954499f7abaabaa882bbe"

SRC_URI = "${GNU_MIRROR}/hello/hello-${PV}.tar.gz"

inherit autotools gettext
```

The variable `LIC_FILES_CHKSUM` is used to track source license changes as described in the "[Tracking License Changes](#)" section in the Yocto Project Overview and Concepts Manual. You can quickly create Autotool-based recipes in a manner similar to the previous example.

7.3.21.3. Makefile-Based Package

Applications that use GNU `make` also require a recipe that has the source archive listed in [SRC_URI](#). You do not need to add a `do_compile` step since by default BitBake starts the `make` command to compile the application. If you need additional `make` options, you should store them in the `EXTRA_OEMAKE` or `PACKAGECONFIG_CONFARGS` variables. BitBake passes these options into the GNU `make` invocation. Note that a `do_install` task is still required. Otherwise, BitBake runs an empty `do_install` task by default.

Some applications might require extra parameters to be passed to the compiler. For example, the application might need an additional header path. You can accomplish this by adding to the `CFLAGS` variable. The following example shows this:

```
CFLAGS_prepend = "-I ${S}/include "
```

In the following example, `mtdev-utils` is a makefile-based package:

```
SUMMARY = "Tools for managing memory technology devices"
SECTION = "base"
DEPENDS = "zlib lzo e2fsprogs util-linux"
HOMEPAGE = "http://www.linux-mtd.infradead.org/"
LICENSE = "GPLv2+"
LIC_FILES_CHKSUM = "file://COPYING;md5=0636e73ff0215e8d672dc4c32c317bb3 \
    file://include/common.h;beginline=1;endline=17;md5=ba05b07912a44ea2bf81ce409380049c"

# Use the latest version at 26 Oct, 2013
SRCREV = "9f107132a6a073cce37434ca9cda6917dd8d866b"
SRC_URI = "git://git.infradead.org/mtdev-utils.git \
    file://add-exclusion-to-mkfs-jffs2-git-2.patch \
"

PV = "1.5.1+git${SRCPV}"

S = "${WORKDIR}/git"

EXTRA_OEMAKE = "'CC=${CC}' 'RANLIB=${RANLIB}' 'AR=${AR}' 'CFLAGS=${CFLAGS}' -I${S}/include -DWITHOUT_XATTR' 'BUILDDIR=${S}'"

do_install () {
    oe_runmake install DESTDIR=${D} SBINDIR=${sbindir} MANDIR=${mandir} INCLUDEDIR=${includedir}
}

PACKAGES += "mtdev-utils-jffs2 mtdev-utils-ubifs mtdev-utils-misc"

FILES_mtdev-utils-jffs2 = "${sbindir}/mkfs.jffs2 ${sbindir}/jffs2dump ${sbindir}/jffs2reader ${sbindir}/sumtool"
FILES_mtdev-utils-ubifs = "${sbindir}/mkfs.ubifs ${sbindir}/ubi*"
FILES_mtdev-utils-misc = "${sbindir}/nftl* ${sbindir}/ftl* ${sbindir}/rfd* ${sbindir}/doc* ${sbindir}/serve_image ${sbindir}

PARALLEL_MAKE = ""

BBCLASSEXTEND = "native"
```

7.3.21.4. Splitting an Application into Multiple Packages

You can use the variables `PACKAGES` and `FILES` to split an application into multiple packages.

Following is an example that uses the `libxpm` recipe. By default, this recipe generates a single package that contains the library along with a few binaries. You can modify the recipe to split the binaries into separate packages:

```
require xorg-lib-common.inc

SUMMARY = "Xpm: X Pixmap extension library"
LICENSE = "BSD"
LIC_FILES_CHKSUM = "file://COPYING;md5=51f4270b012ecd4ab1a164f5f4ed6cf7"
```

```

DEPENDS += "libxext libsm libxt"
PE = "1"

XORG_PN = "libXpm"

PACKAGES += "sxpmp cxpmp"
FILES_cxpmp = "${bindir}/cxpmp"
FILES_sxpmp = "${bindir}/sxpmp"

```

In the previous example, we want to ship the `sxpmp` and `cxpmp` binaries in separate packages. Since `bindir` would be packaged into the main `PN` package by default, we prepend the `PACKAGES` variable so additional package names are added to the start of list. This results in the extra `FILES_*` variables then containing information that define which files and directories go into which packages. Files included by earlier packages are skipped by latter packages. Thus, the main `PN` package does not include the above listed files.

7.3.21.5. Packaging Externally Produced Binaries¶

Sometimes, you need to add pre-compiled binaries to an image. For example, suppose that binaries for proprietary code exist, which are created by a particular division of a company. Your part of the company needs to use those binaries as part of an image that you are building using the OpenEmbedded build system. Since you only have the binaries and not the source code, you cannot use a typical recipe that expects to fetch the source specified in `SRC_URI` and then compile it.

One method is to package the binaries and then install them as part of the image. Generally, it is not a good idea to package binaries since, among other things, it can hinder the ability to reproduce builds and could lead to compatibility problems with ABI in the future. However, sometimes you have no choice.

The easiest solution is to create a recipe that uses the `bin_package` class and to be sure that you are using default locations for build artifacts. In most cases, the `bin_package` class handles "skipping" the configure and compile steps as well as sets things up to grab packages from the appropriate area. In particular, this class sets `noexec` on both the `do_configure` and `do_compile` tasks, sets `FILES_${PN}` to `"/"` so that it picks up all files, and sets up a `do_install` task, which effectively copies all files from `${S}` to `${D}`. The `bin_package` class works well when the files extracted into `${S}` are already laid out in the way they should be laid out on the target. For more information on these variables, see the `FILES`, `PN`, `S`, and `D` variables in the Yocto Project Reference Manual's variable glossary.

Notes

- Using `DEPENDS` is a good idea even for components distributed in binary form, and is often necessary for shared libraries. For a shared library, listing the library dependencies in `DEPENDS` makes sure that the libraries are available in the staging sysroot when other recipes link against the library, which might be necessary for successful linking.
- Using `DEPENDS` also allows runtime dependencies between packages to be added automatically. See the "[Automatically Added Runtime Dependencies](#)" section in the Yocto Project Overview and Concepts Manual for more information.

If you cannot use the `bin_package` class, you need to be sure you are doing the following:

- Create a recipe where the `do_configure` and `do_compile` tasks do nothing: It is usually sufficient to just not define these tasks in the recipe, because the default implementations do nothing unless a Makefile is found in `${S}`.

If `${S}` might contain a Makefile, or if you inherit some class that replaces `do_configure` and `do_compile` with custom versions, then you can use the `[noexec]` flag to turn the tasks into no-ops, as follows:

```

do_configure[noexec] = "1"
do_compile[noexec] = "1"

```

Unlike [deleting the tasks](#), using the flag preserves the dependency chain from the `do_fetch`, `do_unpack`, and `do_patch` tasks to the `do_install` task.

- Make sure your `do_install` task installs the binaries appropriately.
- Ensure that you set up `FILES` (usually `FILES_${PN}`) to point to the files you have installed, which of course depends on where you have installed them and whether those files are in different locations than the defaults.

7.3.22. Following Recipe Style Guidelines¶

When writing recipes, it is good to conform to existing style guidelines. The [OpenEmbedded Styleguide](#) wiki page provides rough guidelines for preferred recipe style.

It is common for existing recipes to deviate a bit from this style. However, aiming for at least a consistent style is a good idea. Some practices, such as omitting spaces around `=` operators in assignments or ordering recipe components in an erratic way, are widely seen as poor style.

7.3.23. Recipe Syntax¶

Understanding recipe file syntax is important for writing recipes. The following list overviews the basic items that make up a BitBake recipe file. For more complete BitBake syntax descriptions, see the "[Syntax and Operators](#)" chapter of the BitBake User Manual.

- **Variable Assignments and Manipulations:** Variable assignments allow a value to be assigned to a variable. The assignment can be static text or might include the contents of other variables. In addition to the assignment, appending and prepending operations are also supported.

The following example shows some of the ways you can use variables in recipes:

```
S = "${WORKDIR}/postfix-${PV}"
CFLAGS += "-DNO_ASM"
SRC_URI_append = " file://fixup.patch"
```

- **Functions:** Functions provide a series of actions to be performed. You usually use functions to override the default implementation of a task function or to complement a default function (i.e. append or prepend to an existing function). Standard functions use `sh` shell syntax, although access to OpenEmbedded variables and internal methods are also available.

The following is an example function from the `sed` recipe:

```
do_install () {
    autotools_do_install
    install -d ${D}${base_bindir}
    mv ${D}${bindir}/sed ${D}${base_bindir}/sed
    rmdir ${D}${bindir}/
}
```

It is also possible to implement new functions that are called between existing tasks as long as the new functions are not replacing or complementing the default functions. You can implement functions in Python instead of shell. Both of these options are not seen in the majority of recipes.

- **Keywords:** BitBake recipes use only a few keywords. You use keywords to include common functions (`inherit`), load parts of a recipe from other files (`include` and `require`) and export variables to the environment (`export`).

The following example shows the use of some of these keywords:

```
export POSTCONF = "${STAGING_BINDIR}/postconf"
inherit autoconf
require otherfile.inc
```

- **Comments (#):** Any lines that begin with the hash character (`#`) are treated as comment lines and are ignored:

```
# This is a comment
```

This next list summarizes the most important and most commonly used parts of the recipe syntax. For more information on these parts of the syntax, you can reference the [Syntax and Operators](#) chapter in the BitBake User Manual.

- **Line Continuation (\):** Use the backward slash (`\`) character to split a statement over multiple lines. Place the slash character at the end of the line that is to be continued on the next line:

```
VAR = "A really long \
line"
```

Note

You cannot have any characters including spaces or tabs after the slash character.

- **Using Variables (\${VARIABLE}):** Use the `$ { VARIABLE }` syntax to access the contents of a variable:

```
SRC_URI = "${SOURCEFORGE_MIRROR}/libpng/zlib-${PV}.tar.gz"
```

Note

It is important to understand that the value of a variable expressed in this form does not get substituted automatically. The expansion of these expressions happens on-demand later (e.g. usually when a function that makes reference to the variable executes). This behavior ensures that the values are most appropriate for the context in which they are finally used. On the rare occasion that you do need the variable expression to be expanded immediately, you can use the `:=` operator instead of `=` when you make the assignment, but this is not generally needed.

- **Quote All Assignments ("value"):** Use double quotes around values in all variable assignments (e.g. `"value"`). Following is an example:

```
VAR1 = "${OTHERVAR}"
VAR2 = "The version is ${PV}"
```

- **Conditional Assignment (?:=):** Conditional assignment is used to assign a value to a variable, but only when the variable is currently unset. Use the question mark followed by the equal sign (`?:=`) to make a "soft" assignment used for conditional assignment. Typically, "soft" assignments are used in the `local.conf` file for variables that are allowed to come through from the external environment.

Here is an example where `VAR1` is set to "New value" if it is currently empty. However, if `VAR1` has already been set, it remains unchanged:

```
VAR1 ?= "New value"
```

In this next example, `VAR1` is left with the value "Original value":

```
VAR1 = "Original value"
VAR1 ?= "New value"
```

- **Appending (+=):** Use the plus character followed by the equals sign (`+=`) to append values to existing variables.

Note

This operator adds a space between the existing content of the variable and the new content.

Here is an example:

```
SRC_URI += "file://fix-makefile.patch"
```

- **Prepending (=+):** Use the equals sign followed by the plus character (=+) to prepend values to existing variables.

Note

This operator adds a space between the new content and the existing content of the variable.

Here is an example:

```
VAR =+ "Starts"
```

- **Appending (_append):** Use the `_append` operator to append values to existing variables. This operator does not add any additional space. Also, the operator is applied after all the +=, and =+ operators have been applied and after all = assignments have occurred.

The following example shows the space being explicitly added to the start to ensure the appended value is not merged with the existing value:

```
SRC_URI_append = " file://fix-makefile.patch"
```

You can also use the `_append` operator with overrides, which results in the actions only being performed for the specified target or machine:

```
SRC_URI_append_sh4 = " file://fix-makefile.patch"
```

- **Prepending (_prepend):** Use the `_prepend` operator to prepend values to existing variables. This operator does not add any additional space. Also, the operator is applied after all the +=, and =+ operators have been applied and after all = assignments have occurred.

The following example shows the space being explicitly added to the end to ensure the prepended value is not merged with the existing value:

```
CFLAGS_prepend = "-I${S}/myincludes "
```

You can also use the `_prepend` operator with overrides, which results in the actions only being performed for the specified target or machine:

```
CFLAGS_prepend_sh4 = "-I${S}/myincludes "
```

- **Overrides:** You can use overrides to set a value conditionally, typically based on how the recipe is being built. For example, to set the `KBRANCH` variable's value to "standard/base" for any target `MACHINE`, except for `qemuarm` where it should be set to "standard/arm-versatile-926ejs", you would do the following:

```
KBRANCH = "standard/base"
KBRANCH_qemuarm = "standard/arm-versatile-926ejs"
```

Overrides are also used to separate alternate values of a variable in other situations. For example, when setting variables such as `FILES` and `RDEPENDS` that are specific to individual packages produced by a recipe, you should always use an override that specifies the name of the package.

- **Indentation:** Use spaces for indentation rather than tabs. For shell functions, both currently work. However, it is a policy decision of the Yocto Project to use tabs in shell functions. Realize that some layers have a policy to use spaces for all indentation.
- **Using Python for Complex Operations:** For more advanced processing, it is possible to use Python code during variable assignments (e.g. search and replacement on a variable).

You indicate Python code using the `${@python_code}` syntax for the variable assignment:

```
SRC_URI = "ftp://ftp.info-zip.org/pub/infozip/src/zip${@d.getVar('PV',1).replace('.', ' ')}-tgz"
```

- **Shell Function Syntax:** Write shell functions as if you were writing a shell script when you describe a list of actions to take. You should ensure that your script works with a generic `sh` and that it does not require any `bash` or other shell-specific functionality. The same considerations apply to various system utilities (e.g. `sed`, `grep`, `awk`, and so forth) that you might wish to use. If in doubt, you should check with multiple implementations - including those from BusyBox.

7.4. Adding a New Machine¶

Adding a new machine to the Yocto Project is a straightforward process. This section describes how to add machines that are similar to those that the Yocto Project already supports.

Note

Although well within the capabilities of the Yocto Project, adding a totally new architecture might require changes to `gcc/glibc` and to the site information, which is beyond the scope of this manual.

For a complete example that shows how to add a new machine, see the "[Creating a New BSP Layer Using the `bitbake-layers` Script](#)" section in the Yocto Project Board Support Package (BSP) Developer's Guide.

7.4.1. Adding the Machine Configuration File¶

To add a new machine, you need to add a new machine configuration file to the layer's `conf/machine` directory. This configuration file provides details about the device you are adding.

The OpenEmbedded build system uses the root name of the machine configuration file to reference the new machine. For example, given a machine configuration file named `crownbay.conf`, the build system recognizes the machine as "crownbay".

The most important variables you must set in your machine configuration file or include from a lower-level configuration file are as follows:

- `TARGET_ARCH` (e.g. "arm")
- `PREFERRED_PROVIDER_virtual/kernel`
- `MACHINE_FEATURES` (e.g. "apm screen wifi")

You might also need these variables:

- `SERIAL_CONSOLES` (e.g. "115200;ttyS0 115200;ttyS1")
- `KERNEL_IMAGETYPE` (e.g. "zImage")
- `IMAGE_FSTYPES` (e.g. "tar.gz jffs2")

You can find full details on these variables in the reference section. You can leverage existing machine `.conf` files from `meta-yocto-bsp/conf/machine/`.

7.4.2. Adding a Kernel for the Machine¶

The OpenEmbedded build system needs to be able to build a kernel for the machine. You need to either create a new kernel recipe for this machine, or extend an existing kernel recipe. You can find several kernel recipe examples in the Source Directory at `meta/recipes-kernel/linux` that you can use as references.

If you are creating a new kernel recipe, normal recipe-writing rules apply for setting up a `SRC_URI`. Thus, you need to specify any necessary patches and set `S` to point at the source code. You need to create a `do_configure` task that configures the unpacked kernel with a `defconfig` file. You can do this by using a `make defconfig` command or, more commonly, by copying in a suitable `defconfig` file and then running `make oldconfig`. By making use of `inherit kernel` and potentially some of the `linux-*.inc` files, most other functionality is centralized and the defaults of the class normally work well.

If you are extending an existing kernel recipe, it is usually a matter of adding a suitable `defconfig` file. The file needs to be added into a location similar to `defconfig` files used for other machines in a given kernel recipe. A possible way to do this is by listing the file in the `SRC_URI` and adding the machine to the expression in `COMPATIBLE_MACHINE`:

```
COMPATIBLE_MACHINE = '(qemu86|qemumips)'
```

For more information on `defconfig` files, see the "[Changing the Configuration](#)" section in the Yocto Project Linux Kernel Development Manual.

7.4.3. Adding a Formfactor Configuration File¶

A formfactor configuration file provides information about the target hardware for which the image is being built and information that the build system cannot obtain from other sources such as the kernel. Some examples of information contained in a formfactor configuration file include framebuffer orientation, whether or not the system has a keyboard, the positioning of the keyboard in relation to the screen, and the screen resolution.

The build system uses reasonable defaults in most cases. However, if customization is necessary, you need to create a `machconfig` file in the `meta/recipes-bsp/formfactor/files` directory. This directory contains directories for specific machines such as `qemuarm` and `qemu86`. For information about the settings available and the defaults, see the `meta/recipes-bsp/formfactor/files/config` file found in the same area.

Following is an example for "qemuarm" machine:

```
HAVE_TOUCHSCREEN=1
HAVE_KEYBOARD=1

DISPLAY_CAN_ROTATE=0
DISPLAY_ORIENTATION=0
#DISPLAY_WIDTH_PIXELS=640
#DISPLAY_HEIGHT_PIXELS=480
#DISPLAY_BPP=16
DISPLAY_DPI=150
DISPLAY_SUBPIXEL_ORDER=vrgb
```

7.5. Upgrading Recipes¶

Over time, upstream developers publish new versions for software built by layer recipes. It is recommended to keep recipes up-to-date with upstream version releases.

While several methods exist that allow you upgrade a recipe, you might consider checking on the upgrade status of a recipe first. You can do so using the `devtool check-upgrade-status` command. See the "[Checking on the Upgrade Status of a Recipe](#)" section in the Yocto Project Reference Manual for more information.

The remainder of this section describes three ways you can upgrade a recipe. You can use the Automated Upgrade Helper (AUH) to set up automatic version upgrades. Alternatively, you can use `devtool upgrade` to set up semi-automatic version upgrades. Finally, you can manually upgrade a recipe by editing the recipe itself.

7.5.1. Using the Auto Upgrade Helper (AUH)

The AUH utility works in conjunction with the OpenEmbedded build system in order to automatically generate upgrades for recipes based on new versions being published upstream. Use AUH when you want to create a service that performs the upgrades automatically and optionally sends you an email with the results.

AUH allows you to update several recipes with a single use. You can also optionally perform build and integration tests using images with the results saved to your hard drive and emails of results optionally sent to recipe maintainers. Finally, AUH creates Git commits with appropriate commit messages in the layer's tree for the changes made to recipes.

Note

Conditions do exist when you should not use AUH to upgrade recipes and you should instead use either `devtool upgrade` or upgrade your recipes manually:

- When AUH cannot complete the upgrade sequence. This situation usually results because custom patches carried by the recipe cannot be automatically rebased to the new version. In this case, `devtool upgrade` allows you to manually resolve conflicts.
- When for any reason you want fuller control over the upgrade process. For example, when you want special arrangements for testing.

The following steps describe how to set up the AUH utility:

1. **Be Sure the Development Host is Set Up:** You need to be sure that your development host is set up to use the Yocto Project. For information on how to set up your host, see the "[Preparing the Build Host](#)" section.
2. **Make Sure Git is Configured:** The AUH utility requires Git to be configured because AUH uses Git to save upgrades. Thus, you must have Git user and email configured. The following command shows your configurations:

```
$ git config --list
```

If you do not have the user and email configured, you can use the following commands to do so:

```
$ git config --global user.name some_name
$ git config --global user.email username@domain.com
```

3. **Clone the AUH Repository:** To use AUH, you must clone the repository onto your development host. The following command uses Git to create a local copy of the repository on your system:

```
$ git clone git://git.yoctoproject.org/auto-upgrade-helper
Cloning into 'auto-upgrade-helper'...
remote: Counting objects: 768, done.
remote: Compressing objects: 100% (300/300), done.
remote: Total 768 (delta 499), reused 703 (delta 434)
Receiving objects: 100% (768/768), 191.47 KiB | 98.00 KiB/s, done.
Resolving deltas: 100% (499/499), done.
Checking connectivity... done.
```

AUH is not part of the [OpenEmbedded-Core \(OE-Core\)](#) or [Poky](#) repositories.

4. **Create a Dedicated Build Directory:** Run the `oe-init-build-env` script to create a fresh build directory that you use exclusively for running the AUH utility:

```
$ cd ~/poky
$ source oe-init-build-env your_AUH_build_directory
```

Re-using an existing build directory and its configurations is not recommended as existing settings could cause AUH to fail or behave undesirably.

5. **Make Configurations in Your Local Configuration File:** Several settings need to exist in the `local.conf` file in the build directory you just created for AUH. Make these following configurations:

- If you want to enable [Build History](#), which is optional, you need the following lines in the `conf/local.conf` file:

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

With this configuration and a successful upgrade, a build history "diff" file appears in the `upgrade-helper/work/recipe/buildhistory-diff.txt` file found in your build directory.

- If you want to enable testing through the [testimage](#) class, which is optional, you need to have the following set in your `conf/local.conf` file:

```
INHERIT += "testimage"
```

Note

If your distro does not enable by default `ptest`, which Poky does, you need the following in your `local.conf` file:

```
DISTRO_FEATURES_append = " ptest"
```

6. **Optionally Start a `vncserver`:** If you are running in a server without an X11 session, you need to start a `vncserver`:

```
$ vncserver :1
$ export DISPLAY=:1
```

7. **Create and Edit an AUH Configuration File:** You need to have the `upgrade-helper/upgrade-helper.conf` configuration file in your build directory. You can find a sample configuration file in the [AUH source repository](#).

Read through the sample file and make configurations as needed. For example, if you enabled build history in your `local.conf` as described earlier, you must enable it in `upgrade-helper.conf`.

Also, if you are using the default `maintainers.inc` file supplied with Poky and located in `meta-yocto` and you do not set a `"maintainers_whitelist"` or `"global_maintainer_override"` in the `upgrade-helper.conf` configuration, and you specify `"-e all"` on the AUH command-line, the utility automatically sends out emails to all the default maintainers. Please avoid this.

This next set of examples describes how to use the AUH:

- **Upgrading a Specific Recipe:** To upgrade a specific recipe, use the following form:

```
$ upgrade-helper.py recipe_name
```

For example, this command upgrades the `xmodmap` recipe:

```
$ upgrade-helper.py xmodmap
```

- **Upgrading a Specific Recipe to a Particular Version:** To upgrade a specific recipe to a particular version, use the following form:

```
$ upgrade-helper.py recipe_name -t version
```

For example, this command upgrades the `xmodmap` recipe to version 1.2.3:

```
$ upgrade-helper.py xmodmap -t 1.2.3
```

- **Upgrading all Recipes to the Latest Versions and Suppressing Email Notifications:** To upgrade all recipes to their most recent versions and suppress the email notifications, use the following command:

```
$ upgrade-helper.py all
```

- **Upgrading all Recipes to the Latest Versions and Send Email Notifications:** To upgrade all recipes to their most recent versions and send email messages to maintainers for each attempted recipe as well as a status email, use the following command:

```
$ upgrade-helper.py -e all
```

Once you have run the AUH utility, you can find the results in the AUH build directory:

```
${BUILDDIR}/upgrade-helper/timestamp
```

The AUH utility also creates recipe update commits from successful upgrade attempts in the layer tree.

You can easily set up to run the AUH utility on a regular basis by using a cron job. See the [weekly.job.sh](#) file distributed with the utility for an example.

7.5.2. Using `devtool upgrade`

As mentioned earlier, an alternative method for upgrading recipes to newer versions is to use `devtool upgrade`. You can read about `devtool upgrade` in general in the ["Use devtool upgrade to Create a Version of the Recipe that Supports a Newer Version of the Software"](#) section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) Manual.

To see all the command-line options available with `devtool upgrade`, use the following help command:

```
$ devtool upgrade -h
```

If you want to find out what version a recipe is currently at upstream without any attempt to upgrade your local version of the recipe, you can use the following command:

```
$ devtool latest-version recipe_name
```

As mentioned in the previous section describing AUH, `devtool upgrade` works in a less-automated manner than AUH. Specifically, `devtool upgrade` only works on a single recipe that you name on the command line, cannot perform build and integration testing using images, and does not automatically generate commits for changes in the source tree. Despite all these "limitations", `devtool upgrade` updates the recipe file to the new upstream version and attempts to rebase custom patches contained by the recipe as needed.

Note

AUH uses much of `devtool upgrade` behind the scenes making AUH somewhat of a "wrapper" application for `devtool upgrade`.

A typical scenario involves having used Git to clone an upstream repository that you use during build operations. Because you are (or have) built the recipe in the past, the layer is likely added to your configuration already. If for some reason, the layer is not added, you could add it easily using the `bitbake-layers` script. For example, suppose you use the `nano.bb` recipe from the `meta-oe` layer in the `meta-openembedded` repository. For this example, assume that the layer has been cloned into following area:

```
/home/scottrif/meta-openembedded
```

The following command from your `Build Directory` adds the layer to your build configuration (i.e. `${BUILDDIR}/conf/bblayers.conf`):

```
$ bitbake-layers add-layer /home/scottrif/meta-openembedded/meta-oe
NOTE: Starting bitbake server...
Parsing recipes: 100% |#####| Time: 0:00:55
Parsing of 1431 .bb files complete (0 cached, 1431 parsed). 2040 targets, 56 skipped, 0 masked, 0 errors.
Removing 12 recipes from the x86_64 sysroot: 100% |#####| Time: 0:00:00
Removing 1 recipes from the x86_64_i586 sysroot: 100% |#####| Time: 0:00:00
Removing 5 recipes from the i586 sysroot: 100% |#####| Time: 0:00:00
Removing 5 recipes from the qemux86 sysroot: 100% |#####| Time: 0:00:00
```

For this example, assume that the `nano.bb` recipe that is upstream has a 2.9.3 version number. However, the version in the local repository is 2.7.4. The following command from your build directory automatically upgrades the recipe for you:

Note

Using the `-V` option is not necessary. Omitting the version number causes `devtool upgrade` to upgrade the recipe to the most recent version.

```
$ devtool upgrade nano -V 2.9.3
NOTE: Starting bitbake server...
NOTE: Creating workspace layer in /home/scottrif/poky/build/workspace
Parsing recipes: 100% |#####| Time: 0:00:46
Parsing of 1431 .bb files complete (0 cached, 1431 parsed). 2040 targets, 56 skipped, 0 masked, 0 errors.
NOTE: Extracting current version source...
NOTE: Resolving any missing task queue dependencies
.
.
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
NOTE: Tasks Summary: Attempted 74 tasks of which 72 didn't need to be rerun and all succeeded.
Adding changed files: 100% |#####| Time: 0:00:00
NOTE: Upgraded source extracted to /home/scottrif/poky/build/workspace/sources/nano
NOTE: New recipe is /home/scottrif/poky/build/workspace/recipes/nano/nano_2.9.3.bb
```

Continuing with this example, you can use `devtool build` to build the newly upgraded recipe:

```
$ devtool build nano
NOTE: Starting bitbake server...
Loading cache: 100% |#####| Time: 0:00:00
Loaded 2040 entries from dependency cache.
Parsing recipes: 100% |#####| Time: 0:00:00
Parsing of 1432 .bb files complete (1431 cached, 1 parsed). 2041 targets, 56 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
.
.
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
NOTE: nano: compiling from external source tree /home/scottrif/poky/build/workspace/sources/nano
NOTE: Tasks Summary: Attempted 520 tasks of which 304 didn't need to be rerun and all succeeded.
```

Within the `devtool upgrade` workflow, opportunity exists to deploy and test your rebuilt software. For this example, however, running `devtool finish` cleans up the workspace once the source in your workspace is clean. This usually means using Git to stage and submit commits for the changes generated by the upgrade process.

Once the tree is clean, you can clean things up in this example with the following command from the `${BUILDDIR}/workspace/sources/nano` directory:

```
$ devtool finish nano meta-oe
NOTE: Starting bitbake server...
Loading cache: 100% |#####| Time: 0:00:00
Loaded 2040 entries from dependency cache.
Parsing recipes: 100% |#####| Time: 0:00:00
Parsing of 1432 .bb files complete (1431 cached, 1 parsed). 2041 targets, 56 skipped, 0 masked, 0 errors.
NOTE: Adding new patch 0001-nano.bb-Stuff-I-changed-when-upgrading-nano.bb.patch
NOTE: Updating recipe nano_2.9.3.bb
NOTE: Removing file /home/scottrif/meta-openembedded/meta-oe/recipes-support/nano/nano_2.7.4.bb
NOTE: Moving recipe file to /home/scottrif/meta-openembedded/meta-oe/recipes-support/nano
NOTE: Leaving source tree /home/scottrif/poky/build/workspace/sources/nano as-is; if you no longer need it then please de
```

Using the `devtool finish` command cleans up the workspace and creates a patch file based on your commits. The tool puts all patch files back into the source directory in a sub-directory named `nano` in this case.

7.5.3. Manually Upgrading a Recipe¶

If for some reason you choose not to upgrade recipes using the [Auto Upgrade Helper \(AUH\)](#) or by using `devtool upgrade`, you can manually edit the recipe files to upgrade the versions.

Caution

Manually updating multiple recipes scales poorly and involves many steps. The recommendation to upgrade recipe versions is through AUH or `devtool upgrade`, both of which automate some steps and provide guidance for others needed for the manual process.

To manually upgrade recipe versions, follow these general steps:

1. **Change the Version:** Rename the recipe such that the version (i.e. the `PV` part of the recipe name) changes appropriately. If the version is not part of the recipe name, change the value as it is set for `PV` within the recipe itself.
2. **Update `SRCREV` if Needed:** If the source code your recipe builds is fetched from Git or some other version control system, update `SRCREV` to point to the commit hash that matches the new version.
3. **Build the Software:** Try to build the recipe using BitBake. Typical build failures include the following:
 - License statements were updated for the new version. For this case, you need to review any changes to the license and update the values of `LICENSE` and `LIC_FILES_CHKSUM` as needed.

Note

License changes are often inconsequential. For example, the license text's copyright year might have changed.

- Custom patches carried by the older version of the recipe might fail to apply to the new version. For these cases, you need to review the failures. Patches might not be necessary for the new version of the software if the upgraded version has fixed those issues. If a patch is necessary and failing, you need to rebase it into the new version.
4. **Optionally Attempt to Build for Several Architectures:** Once you successfully build the new software for a given architecture, you could test the build for other architectures by changing the `MACHINE` variable and rebuilding the software. This optional step is especially important if the recipe is to be released publicly.
 5. **Check the Upstream Change Log or Release Notes:** Checking both these reveals if new features exist that could break backwards-compatibility. If so, you need to take steps to mitigate or eliminate that situation.
 6. **Optionally Create a Bootable Image and Test:** If you want, you can test the new software by booting it onto actual hardware.
 7. **Create a Commit with the Change in the Layer Repository:** After all builds work and any testing is successful, you can create commits for any changes in the layer holding your upgraded recipe.

7.6. Finding Temporary Source Code¶

You might find it helpful during development to modify the temporary source code used by recipes to build packages. For example, suppose you are developing a patch and you need to experiment a bit to figure out your solution. After you have initially built the package, you can iteratively tweak the source code, which is located in the [Build Directory](#), and then you can force a re-compile and quickly test your altered code. Once you settle on a solution, you can then preserve your changes in the form of patches.

During a build, the unpacked temporary source code used by recipes to build packages is available in the Build Directory as defined by the `S` variable. Below is the default value for the `S` variable as defined in the `meta/conf/bitbake.conf` configuration file in the [Source Directory](#):

```
S = "${WORKDIR}/${BP}"
```

You should be aware that many recipes override the `S` variable. For example, recipes that fetch their source from Git usually set `S` to `${WORKDIR}/git`.

Note

The `BP` represents the base recipe name, which consists of the name and version:

```
BP = "${BPN}-${PV}"
```

The path to the work directory for the recipe (`WORKDIR`) is defined as follows:

```
${TMPDIR}/work/${MULTIMACH_TARGET_SYS}/${PN}/${EXTENDPE}${PV}-${PR}
```

The actual directory depends on several things:

- `TMPDIR`: The top-level build output directory.

- `MULTIMACH_TARGET_SYS`: The target system identifier.
- `PN`: The recipe name.
- `EXTENDPE`: The epoch - (if `PE` is not specified, which is usually the case for most recipes, then `EXTENDPE` is blank).
- `PV`: The recipe version.
- `PR`: The recipe revision.

As an example, assume a Source Directory top-level folder named `poky`, a default Build Directory at `poky/build`, and a `qemux86-poky-linux` machine target system. Furthermore, suppose your recipe is named `foo_1.3.0.bb`. In this case, the work directory the build system uses to build the package would be as follows:

```
poky/build/tmp/work/qemux86-poky-linux/foo/1.3.0-r0
```

7.7. Using Quilt in Your Workflow¶

[Quilt](#) is a powerful tool that allows you to capture source code changes without having a clean source tree. This section outlines the typical workflow you can use to modify source code, test changes, and then preserve the changes in the form of a patch all using Quilt.

Tip

With regard to preserving changes to source files, if you clean a recipe or have `rm_work` enabled, the [devtool workflow](#) as described in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual is a safer development flow than the flow that uses Quilt.

Follow these general steps:

1. **Find the Source Code:** Temporary source code used by the OpenEmbedded build system is kept in the [Build Directory](#). See the ["Finding Temporary Source Code"](#) section to learn how to locate the directory that has the temporary source code for a particular package.
2. **Change Your Working Directory:** You need to be in the directory that has the temporary source code. That directory is defined by the `S` variable.
3. **Create a New Patch:** Before modifying source code, you need to create a new patch. To create a new patch file, use `quilt new` as below:

```
$ quilt new my_changes.patch
```

4. **Notify Quilt and Add Files:** After creating the patch, you need to notify Quilt about the files you plan to edit. You notify Quilt by adding the files to the patch you just created:

```
$ quilt add file1.c file2.c file3.c
```

5. **Edit the Files:** Make your changes in the source code to the files you added to the patch.
6. **Test Your Changes:** Once you have modified the source code, the easiest way to test your changes is by calling the `do_compile` task as shown in the following example:

```
$ bitbake -c compile -f package
```

The `-f` or `--force` option forces the specified task to execute. If you find problems with your code, you can just keep editing and re-testing iteratively until things work as expected.

Note

All the modifications you make to the temporary source code disappear once you run the `do_clean` or `do_cleanall` tasks using BitBake (i.e. `bitbake -c clean package` and `bitbake -c cleanall package`). Modifications will also disappear if you use the `rm_work` feature as described in the ["Conserving Disk Space During Builds"](#) section.

7. **Generate the Patch:** Once your changes work as expected, you need to use Quilt to generate the final patch that contains all your modifications.

```
$ quilt refresh
```

At this point, the `my_changes.patch` file has all your edits made to the `file1.c`, `file2.c`, and `file3.c` files.

You can find the resulting patch file in the `patches/` subdirectory of the source (`S`) directory.

8. **Copy the Patch File:** For simplicity, copy the patch file into a directory named `files`, which you can create in the same directory that holds the recipe (`.bb`) file or the append (`.bbappend`) file. Placing the patch here guarantees that the OpenEmbedded build system will find the patch. Next, add the patch into the `SRC_URI` of the recipe. Here is an example:

```
SRC_URI += "file://my_changes.patch"
```

7.8. Using a Development Shell¶

When debugging certain commands or even when just editing packages, `devshell` can be a useful tool. When you invoke `devshell`, all tasks up to and including `do_patch` are run for the specified target. Then, a new terminal is opened and you are placed in `${S}`, the source directory. In the new terminal, all the OpenEmbedded build-related environment variables are still defined so you can use commands such as `configure` and `make`. The commands execute just as if the OpenEmbedded build system were executing them. Consequently, working this way can be helpful when debugging a build or preparing software to be used with the OpenEmbedded build system.

Following is an example that uses `devshell` on a target named `matchbox-desktop`:

```
$ bitbake matchbox-desktop -c devshell
```

This command spawns a terminal with a shell prompt within the OpenEmbedded build environment. The `OE_TERMINAL` variable controls what type of shell is opened.

For spawned terminals, the following occurs:

- The `PATH` variable includes the cross-toolchain.
- The `pkgconfig` variables find the correct `.pc` files.
- The `configure` command finds the Yocto Project site files as well as any other necessary files.

Within this environment, you can run `configure` or `compile` commands as if they were being run by the OpenEmbedded build system itself. As noted earlier, the working directory also automatically changes to the Source Directory (`S`).

To manually run a specific task using `devshell`, run the corresponding `run.*` script in the `${WORKDIR}/temp` directory (e.g., `run.do_configure.pid`). If a task's script does not exist, which would be the case if the task was skipped by way of the `sstate` cache, you can create the task by first running it outside of the `devshell`:

```
$ bitbake -c task
```

Notes

- Execution of a task's `run.*` script and BitBake's execution of a task are identical. In other words, running the script re-runs the task just as it would be run using the `bitbake -c` command.
- Any `run.*` file that does not have a `.pid` extension is a symbolic link (symlink) to the most recent version of that file.

Remember, that the `devshell` is a mechanism that allows you to get into the BitBake task execution environment. And as such, all commands must be called just as BitBake would call them. That means you need to provide the appropriate options for cross-compilation and so forth as applicable.

When you are finished using `devshell`, exit the shell or close the terminal window.

Notes

- It is worth remembering that when using `devshell` you need to use the full compiler name such as `arm-poky-linux-gnueabi-gcc` instead of just using `gcc`. The same applies to other applications such as `binutils`, `libtool` and so forth. BitBake sets up environment variables such as `CC` to assist applications, such as `make` to find the correct tools.
- It is also worth noting that `devshell` still works over X11 forwarding and similar situations.

7.9. Using a Development Python Shell¶

Similar to working within a development shell as described in the previous section, you can also spawn and work within an interactive Python development shell. When debugging certain commands or even when just editing packages, `devpyshell` can be a useful tool. When you invoke `devpyshell`, all tasks up to and including `do_patch` are run for the specified target. Then a new terminal is opened. Additionally, key Python objects and code are available in the same way they are to BitBake tasks, in particular, the data store 'd'. So, commands such as the following are useful when exploring the data store and running functions:

```
pydevshell> d.getVar("STAGING_DIR")
'/media/build1/poky/build/tmp/sysroots'
pydevshell> d.getVar("STAGING_DIR")
'${TMPDIR}/sysroots'
pydevshell> d.setVar("FOO", "bar")
pydevshell> d.getVar("FOO")
'bar'
pydevshell> d.delVar("FOO")
pydevshell> d.getVar("FOO")
pydevshell> bb.build.exec_func("do_unpack", d)
pydevshell>
```

The commands execute just as if the OpenEmbedded build system were executing them. Consequently, working this way can be helpful when debugging a build or preparing software to be used with the OpenEmbedded build system.

Following is an example that uses `devpyshell` on a target named `matchbox-desktop`:

```
$ bitbake matchbox-desktop -c devpyshell
```

This command spawns a terminal and places you in an interactive Python interpreter within the OpenEmbedded build environment. The `OE_TERMINAL` variable controls what type of shell is opened.

When you are finished using `devpyshell`, you can exit the shell either by using `Ctrl+d` or closing the terminal window.

7.10. Building¶

This section describes various build procedures. For example, the steps needed for a simple build, a target that uses multiple configurations, building an image for more than one machine, and so forth.

7.10.1. Building a Simple Image¶

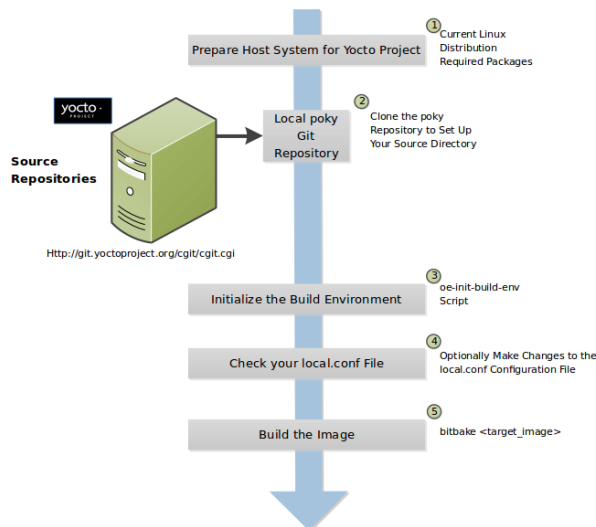
In the development environment, you need to build an image whenever you change hardware support, add or change system libraries, or add or change services that have dependencies. Several methods exist that allow you to build an image within the Yocto Project. This section presents the basic steps you need to build a simple image using BitBake from a build host running Linux.

Notes

- For information on how to build an image using `Toaster`, see the `Toaster User Manual`.
- For information on how to use `devtool` to build images, see the "[Using devtool in Your SDK Workflow](#)" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.
- For a quick example on how to build an image using the OpenEmbedded build system, see the Yocto Project Quick Build document.

The build process creates an entire Linux distribution from source and places it in your `Build Directory` under `tmp/depoy/images`. For detailed information on the build process using BitBake, see the "[Images](#)" section in the Yocto Project Overview and Concepts Manual.

The following figure and list overviews the build process:



- 1. Set up Your Host Development System to Support Development Using the Yocto Project:** See the "[Setting Up to Use the Yocto Project](#)" section for options on how to get a build host ready to use the Yocto Project.
- 2. Initialize the Build Environment:** Initialize the build environment by sourcing the build environment script (i.e. `oe-init-build-env`):

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

When you use the initialization script, the OpenEmbedded build system uses `build` as the default Build Directory in your current work directory. You can use a `build_dir` argument with the script to specify a different build directory.

Tip

A common practice is to use a different Build Directory for different targets. For example, `~/build/x86` for a `qemux86` target, and `~/build/arm` for a `qemuarm` target.

- 3. Make Sure Your `local.conf` File is Correct:** Ensure the `conf/local.conf` configuration file, which is found in the Build Directory, is set up how you want it. This file defines many aspects of the build environment including the target machine architecture through the `MACHINE` variable, the packaging format used during the build (`PACKAGE_CLASSES`), and a centralized tarball download directory through the `DL_DIR` variable.

4. **Build the Image:** Build the image using the `bitbake` command:

```
$ bitbake target
```

Note

For information on BitBake, see the BitBake User Manual.

The *target* is the name of the recipe you want to build. Common targets are the images in `meta/recipes-core/images`, `meta/recipes-sato/images`, and so forth all found in the [Source Directory](#). Or, the target can be the name of a recipe for a specific piece of software such as BusyBox. For more details about the images the OpenEmbedded build system supports, see the "[Images](#)" chapter in the Yocto Project Reference Manual.

As an example, the following command builds the `core-image-minimal` image:

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

Once an image has been built, it often needs to be installed. The images and kernels built by the OpenEmbedded build system are placed in the Build Directory in `tmp/deploy/images`. For information on how to run pre-built images such as `qemux86` and `qemuarm`, see the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual. For information about how to install these images, see the documentation for your particular board or machine.

7.10.2. Building Images for Multiple Targets Using Multiple Configurations¶

You can use a single `bitbake` command to build multiple images or packages for different targets where each image or package requires a different configuration (multiple configuration builds). The builds, in this scenario, are sometimes referred to as "multiconfigs", and this section uses that term throughout.

This section describes how to set up for multiple configuration builds and how to account for cross-build dependencies between the multiconfigs.

7.10.2.1. Setting Up and Running a Multiple Configuration Build¶

To accomplish a multiple configuration build, you must define each target's configuration separately using a parallel configuration file in the [Build Directory](#), and you must follow a required file hierarchy. Additionally, you must enable the multiple configuration builds in your `local.conf` file.

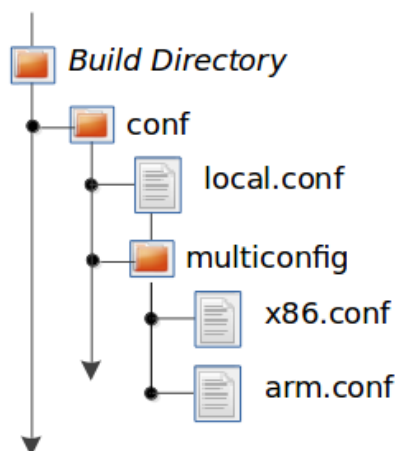
Follow these steps to set up and execute multiple configuration builds:

- **Create Separate Configuration Files:** You need to create a single configuration file for each build target (each multiconfig). Minimally, each configuration file must define the machine and the temporary directory BitBake uses for the build. Suggested practice dictates that you do not overlap the temporary directories used during the builds. However, it is possible that you can share the temporary directory ([TMPDIR](#)). For example, consider a scenario with two different multiconfigs for the same [MACHINE](#): "qemux86" built for two distributions such as "poky" and "poky-lsb". In this case, you might want to use the same `TMPDIR`.

Here is an example showing the minimal statements needed in a configuration file for a "qemux86" target whose temporary build directory is `tmpmultix86`:

```
MACHINE="qemux86"
TMPDIR="${TOPDIR}/tmpmultix86"
```

The location for these multiconfig configuration files is specific. They must reside in the current build directory in a sub-directory of `conf` named `multiconfig`. Following is an example that defines two configuration files for the "x86" and "arm" multiconfigs:



The reason for this required file hierarchy is because the `BBPATH` variable is not constructed until the layers are parsed. Consequently, using the configuration file as a pre-configuration file is not possible unless it is located in the current working directory.

- **Add the BitBake Multi-configuration Variable to the Local Configuration File:** Use the `BBMULTICONFIG` variable in your `conf/local.conf` configuration file to specify each multiconfig. Continuing with the example from the previous figure, the

BBMULTICONFIG variable needs to enable two multiconfigs: "x86" and "arm" by specifying each configuration file:

```
BBMULTICONFIG = "x86 arm"
```

Note

A "default" configuration already exists by definition. This configuration is named: "" (i.e. empty string) and is defined by the variables coming from your `local.conf` file. Consequently, the previous example actually adds two additional configurations to your build: "arm" and "x86" along with "".

- **Launch BitBake:** Use the following BitBake command form to launch the multiple configuration build:

```
$ bitbake [mc:multiconfigname:]target [[mc:multiconfigname:]target] ... ]
```

For the example in this section, the following command applies:

```
$ bitbake mc:x86:core-image-minimal mc:arm:core-image-sato mc::core-image-base
```

The previous BitBake command builds a `core-image-minimal` image that is configured through the `x86.conf` configuration file, a `core-image-sato` image that is configured through the `arm.conf` configuration file and a `core-image-base` that is configured through your `local.conf` configuration file.

Note

Support for multiple configuration builds in the Yocto Project 3.0.1 (Zeus) Release does not include Shared State (sstate) optimizations. Consequently, if a build uses the same object twice in, for example, two different `TMPDIR` directories, the build either loads from an existing sstate cache for that build at the start or builds the object fresh.

7.10.2.2. Enabling Multiple Configuration Build Dependencies¶

Sometimes dependencies can exist between targets (multiconfigs) in a multiple configuration build. For example, suppose that in order to build a `core-image-sato` image for an "x86" multiconfig, the root filesystem of an "arm" multiconfig must exist. This dependency is essentially that the `do_image` task in the `core-image-sato` recipe depends on the completion of the `do_rootfs` task of the `core-image-minimal` recipe.

To enable dependencies in a multiple configuration build, you must declare the dependencies in the recipe using the following statement form:

```
task_or_package[mcdepends] = "mc:from_multiconfig:to_multiconfig:recipe_name:task_on_which_to_depend"
```

To better show how to use this statement, consider the example scenario from the first paragraph of this section. The following statement needs to be added to the recipe that builds the `core-image-sato` image:

```
do_image[mcdepends] = "mc:x86:arm:core-image-minimal:do_rootfs"
```

In this example, the `from_multiconfig` is "x86". The `to_multiconfig` is "arm". The task on which the `do_image` task in the recipe depends is the `do_rootfs` task from the `core-image-minimal` recipe associated with the "arm" multiconfig.

Once you set up this dependency, you can build the "x86" multiconfig using a BitBake command as follows:

```
$ bitbake mc:x86:core-image-sato
```

This command executes all the tasks needed to create the `core-image-sato` image for the "x86" multiconfig. Because of the dependency, BitBake also executes through the `do_rootfs` task for the "arm" multiconfig build.

Having a recipe depend on the root filesystem of another build might not seem that useful. Consider this change to the statement in the `core-image-sato` recipe:

```
do_image[mcdepends] = "mc:x86:arm:core-image-minimal:do_image"
```

In this case, BitBake must create the `core-image-minimal` image for the "arm" build since the "x86" build depends on it.

Because "x86" and "arm" are enabled for multiple configuration builds and have separate configuration files, BitBake places the artifacts for each build in the respective temporary build directories (i.e. `TMPDIR`).

7.10.3. Building an Initial RAM Filesystem (initramfs) Image¶

An initial RAM filesystem (initramfs) image provides a temporary root filesystem used for early system initialization (e.g. loading of modules needed to locate and mount the "real" root filesystem).

Note

The initramfs image is the successor of initial RAM disk (initrd). It is a "copy in and out" (cpio) archive of the initial filesystem that gets loaded into memory during the Linux startup process. Because Linux uses the contents of the archive during initialization, the initramfs image needs to contain all of the device drivers and tools needed to mount the final root filesystem.

Follow these steps to create an initramfs image:

1. **Create the *initramfs Image Recipe*:** You can reference the `core-image-minimal-initramfs.bb` recipe found in the `meta/recipes-core` directory of the [Source Directory](#) as an example from which to work.
2. **Decide if You Need to Bundle the *initramfs Image Into the Kernel Image*:** If you want the *initramfs* image that is built to be bundled in with the kernel image, set the `INITRAMFS_IMAGE_BUNDLE` variable to "1" in your `local.conf` configuration file and set the `INITRAMFS_IMAGE` variable in the recipe that builds the kernel image.

Tip

It is recommended that you do bundle the *initramfs* image with the kernel image to avoid circular dependencies between the kernel recipe and the *initramfs* recipe should the *initramfs* image include kernel modules.

Setting the `INITRAMFS_IMAGE_BUNDLE` flag causes the *initramfs* image to be unpacked into the `${B}/usr/` directory. The unpacked *initramfs* image is then passed to the kernel's `Makefile` using the `CONFIG_INITRAMFS_SOURCE` variable, allowing the *initramfs* image to be built into the kernel normally.

Note

If you choose to not bundle the *initramfs* image with the kernel image, you are essentially using an [Initial RAM Disk \(initrd\)](#). Creating an *initrd* is handled primarily through the `INITRD_IMAGE`, `INITRD_LIVE`, and `INITRD_IMAGE_LIVE` variables. For more information, see the [image-live.bbclass](#) file.

3. **Optionally Add Items to the *initramfs Image Through the initramfs Image Recipe*:** If you add items to the *initramfs* image by way of its recipe, you should use `PACKAGE_INSTALL` rather than `IMAGE_INSTALL`. `PACKAGE_INSTALL` gives more direct control of what is added to the image as compared to the defaults you might not necessarily want that are set by the `image` or `core-image` classes.
4. **Build the Kernel Image and the *initramfs Image*:** Build your kernel image using BitBake. Because the *initramfs* image recipe is a dependency of the kernel image, the *initramfs* image is built as well and bundled with the kernel image if you used the `INITRAMFS_IMAGE_BUNDLE` variable described earlier.

7.10.4. Building a Tiny System¶

Very small distributions have some significant advantages such as requiring less on-die or in-package memory (cheaper), better performance through efficient cache usage, lower power requirements due to less memory, faster boot times, and reduced development overhead. Some real-world examples where a very small distribution gives you distinct advantages are digital cameras, medical devices, and small headless systems.

This section presents information that shows you how you can trim your distribution to even smaller sizes than the `poky-tiny` distribution, which is around 5 Mbytes, that can be built out-of-the-box using the Yocto Project.

7.10.4.1. Overview¶

The following list presents the overall steps you need to consider and perform to create distributions with smaller root filesystems, achieve faster boot times, maintain your critical functionality, and avoid initial RAM disks:

- [Determine your goals and guiding principles.](#)
- [Understand what contributes to your image size.](#)
- [Reduce the size of the root filesystem.](#)
- [Reduce the size of the kernel.](#)
- [Eliminate packaging requirements.](#)
- [Look for other ways to minimize size.](#)
- [Iterate on the process.](#)

7.10.4.2. Goals and Guiding Principles¶

Before you can reach your destination, you need to know where you are going. Here is an example list that you can use as a guide when creating very small distributions:

- Determine how much space you need (e.g. a kernel that is 1 Mbyte or less and a root filesystem that is 3 Mbytes or less).
- Find the areas that are currently taking 90% of the space and concentrate on reducing those areas.
- Do not create any difficult "hacks" to achieve your goals.
- Leverage the device-specific options.
- Work in a separate layer so that you keep changes isolated. For information on how to create layers, see the ["Understanding and Creating Layers"](#) section.

7.10.4.3. Understand What Contributes to Your Image Size¶

It is easiest to have something to start with when creating your own distribution. You can use the Yocto Project out-of-the-box to create the `poky-tiny` distribution. Ultimately, you will want to make changes in your own distribution that are likely modeled after `poky-tiny`.

Note

To use `poky-tiny` in your build, set the `DISTRO` variable in your `local.conf` file to "poky-tiny" as described in the "[Creating Your Own Distribution](#)" section.

Understanding some memory concepts will help you reduce the system size. Memory consists of static, dynamic, and temporary memory. Static memory is the TEXT (code), DATA (initialized data in the code), and BSS (uninitialized data) sections. Dynamic memory represents memory that is allocated at runtime: stacks, hash tables, and so forth. Temporary memory is recovered after the boot process. This memory consists of memory used for decompressing the kernel and for the `__init__` functions.

To help you see where you currently are with kernel and root filesystem sizes, you can use two tools found in the [Source Directory](#) in the `scripts/tiny/` directory:

- `ksize.py`: Reports component sizes for the kernel build objects.
- `dirsize.py`: Reports component sizes for the root filesystem.

This next tool and command help you organize configuration fragments and view file dependencies in a human-readable form:

- `merge_config.sh`: Helps you manage configuration files and fragments within the kernel. With this tool, you can merge individual configuration fragments together. The tool allows you to make overrides and warns you of any missing configuration options. The tool is ideal for allowing you to iterate on configurations, create minimal configurations, and create configuration files for different machines without having to duplicate your process.

The `merge_config.sh` script is part of the Linux Yocto kernel Git repositories (i.e. `linux-yocto-3.14`, `linux-yocto-3.10`, `linux-yocto-3.8`, and so forth) in the `scripts/kconfig` directory.

For more information on configuration fragments, see the "[Creating Configuration Fragments](#)" section in the Yocto Project Linux Kernel Development Manual.

- `bitbake -u taskexp -g bitbake_target`: Using the BitBake command with these options brings up a Dependency Explorer from which you can view file dependencies. Understanding these dependencies allows you to make informed decisions when cutting out various pieces of the kernel and root filesystem.

7.10.4.4. Trim the Root Filesystem¶

The root filesystem is made up of packages for booting, libraries, and applications. To change things, you can configure how the packaging happens, which changes the way you build them. You can also modify the filesystem itself or select a different filesystem.

First, find out what is hogging your root filesystem by running the `dirsize.py` script from your root directory:

```
$ cd root-directory-of-image
$ dirsize.py 100000 > dirsize-100k.log
$ cat dirsize-100k.log
```

You can apply a filter to the script to ignore files under a certain size. The previous example filters out any files below 100 Kbytes. The sizes reported by the tool are uncompressed, and thus will be smaller by a relatively constant factor in a compressed root filesystem. When you examine your log file, you can focus on areas of the root filesystem that take up large amounts of memory.

You need to be sure that what you eliminate does not cripple the functionality you need. One way to see how packages relate to each other is by using the Dependency Explorer UI with the BitBake command:

```
$ cd image-directory
$ bitbake -u taskexp -g image
```

Use the interface to select potential packages you wish to eliminate and see their dependency relationships.

When deciding how to reduce the size, get rid of packages that result in minimal impact on the feature set. For example, you might not need a VGA display. Or, you might be able to get by with `devtmpfs` and `mdev` instead of `udev`.

Use your `local.conf` file to make changes. For example, to eliminate `udev` and `glib`, set the following in the local configuration file:

```
VIRTUAL-RUNTIME_dev_manager = ""
```

Finally, you should consider exactly the type of root filesystem you need to meet your needs while also reducing its size. For example, consider `cramfs`, `squashfs`, `ubifs`, `ext2`, or an `initramfs` using `initramfs`. Be aware that `ext3` requires a 1 Mbyte journal. If you are okay with running read-only, you do not need this journal.

Note

After each round of elimination, you need to rebuild your system and then use the tools to see the effects of your reductions.

7.10.4.5. Trim the Kernel¶

The kernel is built by including policies for hardware-independent aspects. What subsystems do you enable? For what architecture are you building? Which drivers do you build by default?

Note

You can modify the kernel source if you want to help with boot time.

Run the `ksize.py` script from the top-level Linux build directory to get an idea of what is making up the kernel:

```
$ cd top-level-linux-build-directory
$ ksize.py > ksize.log
$ cat ksize.log
```

When you examine the log, you will see how much space is taken up with the built-in `.o` files for drivers, networking, core kernel files, filesystem, sound, and so forth. The sizes reported by the tool are uncompressed, and thus will be smaller by a relatively constant factor in a compressed kernel image. Look to reduce the areas that are large and taking up around the "90% rule."

To examine, or drill down, into any particular area, use the `-d` option with the script:

```
$ ksize.py -d > ksize.log
```

Using this option breaks out the individual file information for each area of the kernel (e.g. drivers, networking, and so forth).

Use your log file to see what you can eliminate from the kernel based on features you can let go. For example, if you are not going to need sound, you do not need any drivers that support sound.

After figuring out what to eliminate, you need to reconfigure the kernel to reflect those changes during the next build. You could run `menuconfig` and make all your changes at once. However, that makes it difficult to see the effects of your individual eliminations and also makes it difficult to replicate the changes for perhaps another target device. A better method is to start with no configurations using `allnoconfig`, create configuration fragments for individual changes, and then manage the fragments into a single configuration file using `merge_config.sh`. The tool makes it easy for you to iterate using the configuration change and build cycle.

Each time you make configuration changes, you need to rebuild the kernel and check to see what impact your changes had on the overall size.

7.10.4.6. Remove Package Management Requirements¶

Packaging requirements add size to the image. One way to reduce the size of the image is to remove all the packaging requirements from the image. This reduction includes both removing the package manager and its unique dependencies as well as removing the package management data itself.

To eliminate all the packaging requirements for an image, be sure that "package-management" is not part of your `IMAGE_FEATURES` statement for the image. When you remove this feature, you are removing the package manager as well as its dependencies from the root filesystem.

7.10.4.7. Look for Other Ways to Minimize Size¶

Depending on your particular circumstances, other areas that you can trim likely exist. The key to finding these areas is through tools and methods described here combined with experimentation and iteration. Here are a couple of areas to experiment with:

- `glibc`: In general, follow this process:
 1. Remove `glibc` features from `DISTRO_FEATURES` that you think you do not need.
 2. Build your distribution.
 3. If the build fails due to missing symbols in a package, determine if you can reconfigure the package to not need those features. For example, change the configuration to not support wide character support as is done for `ncurses`. Or, if support for those characters is needed, determine what `glibc` features provide the support and restore the configuration.
 4. Rebuild and repeat the process.
- `busybox`: For BusyBox, use a process similar as described for `glibc`. A difference is you will need to boot the resulting system to see if you are able to do everything you expect from the running system. You need to be sure to integrate configuration fragments into Busybox because BusyBox handles its own core features and then allows you to add configuration fragments on top.

7.10.4.8. Iterate on the Process¶

If you have not reached your goals on system size, you need to iterate on the process. The process is the same. Use the tools and see just what is taking up 90% of the root filesystem and the kernel. Decide what you can eliminate without limiting your device beyond what you need.

Depending on your system, a good place to look might be Busybox, which provides a stripped down version of Unix tools in a single, executable file. You might be able to drop virtual terminal services or perhaps `ipv6`.

7.10.5. Building Images for More than One Machine¶

A common scenario developers face is creating images for several different machines that use the same software environment. In this situation, it is tempting to set the tunings and optimization flags for each build specifically for the targeted hardware (i.e. "maxing out" the tunings). Doing so can considerably add to build times and package feed maintenance collectively for the machines. For example, selecting tunes that are extremely specific to a CPU core used in a system might enable some micro optimizations in GCC for that particular system but would otherwise not gain you much of a performance difference across the other systems as compared to using a more general tuning across all the builds (e.g. setting `DEFAULTTUNE` specifically for each machine's build). Rather than "max out" each build's tunings, you can take steps that cause the OpenEmbedded build system to reuse software across the various machines where it makes sense.

If build speed and package feed maintenance are considerations, you should consider the points in this section that can help you optimize your tunings to best consider build times and package feed maintenance.

- **Share the Build Directory:** If at all possible, share the `TMPDIR` across builds. The Yocto Project supports switching between different `MACHINE` values in the same `TMPDIR`. This practice is well supported and regularly used by developers when building for multiple machines. When you use the same `TMPDIR` for multiple machine builds, the OpenEmbedded build system can reuse the existing native and often cross-recipes for multiple machines. Thus, build time decreases.

Note

If `DISTRO` settings change or fundamental configuration settings such as the filesystem layout, you need to work with a clean `TMPDIR`. Sharing `TMPDIR` under these circumstances might work but since it is not guaranteed, you should use a clean `TMPDIR`.

- **Enable the Appropriate Package Architecture:** By default, the OpenEmbedded build system enables three levels of package architectures: "all", "tune" or "package", and "machine". Any given recipe usually selects one of these package architectures (types) for its output. Depending for what a given recipe creates packages, making sure you enable the appropriate package architecture can directly impact the build time.

A recipe that just generates scripts can enable "all" architecture because there are no binaries to build. To specifically enable "all" architecture, be sure your recipe inherits the `allarch` class. This class is useful for "all" architectures because it configures many variables so packages can be used across multiple architectures.

If your recipe needs to generate packages that are machine-specific or when one of the build or runtime dependencies is already machine-architecture dependent, which makes your recipe also machine-architecture dependent, make sure your recipe enables the "machine" package architecture through the `MACHINE_ARCH` variable:

```
PACKAGE_ARCH = "${MACHINE_ARCH}"
```

When you do not specifically enable a package architecture through the `PACKAGE_ARCH`, The OpenEmbedded build system defaults to the `TUNE_PKGARCH` setting:

```
PACKAGE_ARCH = "${TUNE_PKGARCH}"
```

- **Choose a Generic Tuning File if Possible:** Some tunes are more generic and can run on multiple targets (e.g. an `armv5` set of packages could run on `armv6` and `armv7` processors in most cases). Similarly, `i486` binaries could work on `i586` and higher processors. You should realize, however, that advances on newer processor versions would not be used.

If you select the same tune for several different machines, the OpenEmbedded build system reuses software previously built, thus speeding up the overall build time. Realize that even though a new sysroot for each machine is generated, the software is not recompiled and only one package feed exists.

- **Manage Granular Level Packaging:** Sometimes cases exist where injecting another level of package architecture beyond the three higher levels noted earlier can be useful. For example, consider how NXP (formerly Freescale) allows for the easy reuse of binary packages in their layer `meta-freescale`. In this example, the `fsl-dynamic-packagearch` class shares GPU packages for i.MX53 boards because all boards share the AMD GPU. The i.MX6-based boards can do the same because all boards share the Vivante GPU. This class inspects the BitBake datastore to identify if the package provides or depends on one of the sub-architecture values. If so, the class sets the `PACKAGE_ARCH` value based on the `MACHINE_SUBARCH` value. If the package does not provide or depend on one of the sub-architecture values but it matches a value in the machine-specific filter, it sets `MACHINE_ARCH`. This behavior reduces the number of packages built and saves build time by reusing binaries.
- **Use Tools to Debug Issues:** Sometimes you can run into situations where software is being rebuilt when you think it should not be. For example, the OpenEmbedded build system might not be using shared state between machines when you think it should be. These types of situations are usually due to references to machine-specific variables such as `MACHINE`, `SERIAL_CONSOLES`, `XSERVER`, `MACHINE_FEATURES`, and so forth in code that is supposed to only be tune-specific or when the recipe depends (`DEPENDS`, `RDEPENDS`, `RRECOMMENDS`, `RSUGGESTS`, and so forth) on some other recipe that already has `PACKAGE_ARCH` defined as `"${MACHINE_ARCH}"`.

Note

Patches to fix any issues identified are most welcome as these issues occasionally do occur.

For such cases, you can use some tools to help you sort out the situation:

- **`sstate-diff-machines.sh`:** You can find this tool in the `scripts` directory of the Source Repositories. See the comments in the script for information on how to use the tool.
- **BitBake's "-S printdiff" Option:** Using this option causes BitBake to try to establish the closest signature match it can (e.g. in the shared state cache) and then run `bitbake-diffsigs` over the matches to determine the stamps and delta where these two stamp trees diverge.

7.10.6. Building Software from an External Source¶

By default, the OpenEmbedded build system uses the `Build Directory` when building source code. The build process involves fetching the source files, unpacking them, and then patching them if necessary before the build takes place.

Situations exist where you might want to build software from source files that are external to and thus outside of the OpenEmbedded build system. For example, suppose you have a project that includes a new BSP with a heavily customized kernel. And, you want to minimize exposing the build system to the development team so that they can focus on their project and maintain everyone's workflow as much as possible. In this case, you want a kernel source directory on the development machine where the development occurs. You want the recipe's `SRC_URI` variable to point to the external directory and use it as is, not copy it.

To build from software that comes from an external source, all you need to do is inherit the `externalsrc` class and then set the `EXTERNALSRC` variable to point to your external source code. Here are the statements to put in your `local.conf` file:

```
INHERIT += "externalsrc"
EXTERNALSRC_pn-myrecipe = "path-to-your-source-tree"
```

This next example shows how to accomplish the same thing by setting `EXTERNALSRC` in the recipe itself or in the recipe's append file:

```
EXTERNALSRC = "path"
EXTERNALSRC_BUILD = "path"
```

Note

In order for these settings to take effect, you must globally or locally inherit the `externalsrc` class.

By default, `externalsrc.bbclass` builds the source code in a directory separate from the external source directory as specified by `EXTERNALSRC`. If you need to have the source built in the same directory in which it resides, or some other nominated directory, you can set `EXTERNALSRC_BUILD` to point to that directory:

```
EXTERNALSRC_BUILD_pn-myrecipe = "path-to-your-source-tree"
```

7.10.7. Replicating a Build Offline¶

It can be useful to take a "snapshot" of upstream sources used in a build and then use that "snapshot" later to replicate the build offline. To do so, you need to first prepare and populate your downloads directory your "snapshot" of files. Once your downloads directory is ready, you can use it at any time and from any machine to replicate your build.

Follow these steps to populate your Downloads directory:

1. **Create a Clean Downloads Directory:** Start with an empty downloads directory (`DL_DIR`). You start with an empty downloads directory by either removing the files in the existing directory or by setting `DL_DIR` to point to either an empty location or one that does not yet exist.
2. **Generate Tarballs of the Source Git Repositories:** Edit your `local.conf` configuration file as follows:

```
DL_DIR = "/home/your-download-dir/"
BB_GENERATE_MIRROR_TARBALLS = "1"
```

During the fetch process in the next step, BitBake gathers the source files and creates tarballs in the directory pointed to by `DL_DIR`. See the `BB_GENERATE_MIRROR_TARBALLS` variable for more information.

3. **Populate Your Downloads Directory Without Building:** Use BitBake to fetch your sources but inhibit the build:

```
$ bitbake target --runonly=fetch
```

The downloads directory (i.e. `${DL_DIR}`) now has a "snapshot" of the source files in the form of tarballs, which can be used for the build.

4. **Optionally Remove Any Git or other SCM Subdirectories From the Downloads Directory:** If you want, you can clean up your downloads directory by removing any Git or other Source Control Management (SCM) subdirectories such as `${DL_DIR}/git2/*`. The tarballs already contain these subdirectories.

Once your downloads directory has everything it needs regarding source files, you can create your "own-mirror" and build your target. Understand that you can use the files to build the target offline from any machine and at any time.

Follow these steps to build your target using the files in the downloads directory:

1. **Using Local Files Only:** Inside your `local.conf` file, add the `SOURCE_MIRROR_URL` variable, inherit the `own-mirrors` class, and use the `BB_NO_NETWORK` variable to your `local.conf`.

```
SOURCE_MIRROR_URL ?= "file:///home/your-download-dir/"
INHERIT += "own-mirrors"
BB_NO_NETWORK = "1"
```

The `SOURCE_MIRROR_URL` and `own-mirror` class set up the system to use the downloads directory as your "own mirror". Using the `BB_NO_NETWORK` variable makes sure that BitBake's fetching process in step 3 stays local, which means files from your "own-mirror" are used.

2. **Start With a Clean Build:** You can start with a clean build by removing the `${TMPDIR}` directory or using a new [Build Directory](#).
3. **Build Your Target:** Use BitBake to build your target:

```
$ bitbake target
```

The build completes using the known local "snapshot" of source files from your mirror. The resulting tarballs for your "snapshot" of source files are in the downloads directory.

Note

The offline build does not work if recipes attempt to find the latest version of software by setting `SRCREV` to `${AUTOREV}`:

```
SRCREV = "${AUTOREV}"
```

When a recipe sets `SRCREV` to `${AUTOREV}`, the build system accesses the network in an attempt to determine the latest version of software from the SCM. Typically, recipes that use `AUTOREV` are custom or modified recipes. Recipes that reside in public repositories usually do not use `AUTOREV`.

If you do have recipes that use `AUTOREV`, you can take steps to still use the recipes in an offline build. Do the following:

- a. Use a configuration generated by enabling `build history`.
- b. Use the `buildhistory-collect-srcrevs` command to collect the stored SRCREV values from the build's history. For more information on collecting these values, see the "[Build History Package Information](#)" section.
- c. Once you have the correct source revisions, you can modify those recipes to to set SRCREV to specific versions of the software.

7.11. Speeding Up a Build¶

Build time can be an issue. By default, the build system uses simple controls to try and maximize build efficiency. In general, the default settings for all the following variables result in the most efficient build times when dealing with single socket systems (i.e. a single CPU). If you have multiple CPUs, you might try increasing the default values to gain more speed. See the descriptions in the glossary for each variable for more information:

- `BB_NUMBER_THREADS`: The maximum number of threads BitBake simultaneously executes.
- `BB_NUMBER_PARSE_THREADS`: The number of threads BitBake uses during parsing.
- `PARALLEL_MAKE`: Extra options passed to the `make` command during the `do_compile` task in order to specify parallel compilation on the local build host.
- `PARALLEL_MAKEINST`: Extra options passed to the `make` command during the `do_install` task in order to specify parallel installation on the local build host.

As mentioned, these variables all scale to the number of processor cores available on the build system. For single socket systems, this auto-scaling ensures that the build system fundamentally takes advantage of potential parallel operations during the build based on the build machine's capabilities.

Following are additional factors that can affect build speed:

- File system type: The file system type that the build is being performed on can also influence performance. Using `ext4` is recommended as compared to `ext2` and `ext3` due to `ext4` improved features such as extents.
- Disabling the updating of access time using `noatime`: The `noatime` mount option prevents the build system from updating file and directory access times.
- Setting a longer commit: Using the "`commit=`" mount option increases the interval in seconds between disk cache writes. Changing this interval from the five second default to something longer increases the risk of data loss but decreases the need to write to the disk, thus increasing the build performance.
- Choosing the packaging backend: Of the available packaging backends, IPK is the fastest. Additionally, selecting a singular packaging backend also helps.
- Using `tmpfs` for `TMPDIR` as a temporary file system: While this can help speed up the build, the benefits are limited due to the compiler using `-pipe`. The build system goes to some lengths to avoid `sync()` calls into the file system on the principle that if there was a significant failure, the `Build Directory` contents could easily be rebuilt.
- Inheriting the `rm_work` class: Inheriting this class has shown to speed up builds due to significantly lower amounts of data stored in the data cache as well as on disk. Inheriting this class also makes cleanup of `TMPDIR` faster, at the expense of being easily able to dive into the source code. File system maintainers have recommended that the fastest way to clean up large numbers of files is to reformat partitions rather than delete files due to the linear nature of partitions. This, of course, assumes you structure the disk partitions and file systems in a way that this is practical.

Aside from the previous list, you should keep some trade offs in mind that can help you speed up the build:

- Remove items from `DISTRO_FEATURES` that you might not need.
- Exclude debug symbols and other debug information: If you do not need these symbols and other debug information, disabling the `*-dbg` package generation can speed up the build. You can disable this generation by setting the `INHIBIT_PACKAGE_DEBUG_SPLIT` variable to "1".
- Disable static library generation for recipes derived from `autoconf` or `libtool`: Following is an example showing how to disable static libraries and still provide an override to handle exceptions:

```
STATICLIBCONF = "--disable-static"
STATICLIBCONF_sqlite3-native = ""
EXTRA_OECONF += "${STATICLIBCONF}"
```

Notes

- Some recipes need static libraries in order to work correctly (e.g. `pseudo-native` needs `sqlite3-native`). Overrides, as in the previous example, account for these kinds of exceptions.
- Some packages have packaging code that assumes the presence of the static libraries. If so, you might need to exclude them as well.

7.12. Working With Libraries¶

Libraries are an integral part of your system. This section describes some common practices you might find helpful when working with libraries to build your system:

- [How to include static library files](#)
- [How to use the Multilib feature to combine multiple versions of library files into a single image](#)
- [How to install multiple versions of the same library in parallel on the same system](#)

7.12.1. Including Static Library Files

If you are building a library and the library offers static linking, you can control which static library files (* .a files) get included in the built library.

The `PACKAGES` and `FILES_*` variables in the `meta/conf/bitbake.conf` configuration file define how files installed by the `do_install` task are packaged. By default, the `PACKAGES` variable includes `${PN}-staticdev`, which represents all static library files.

Note

Some previously released versions of the Yocto Project defined the static library files through `${PN}-dev`.

Following is part of the BitBake configuration file, where you can see how the static library files are defined:

```
PACKAGE_BEFORE_PN ?= ""
PACKAGES = "${PN}-dbg ${PN}-staticdev ${PN}-dev ${PN}-doc ${PN}-locale ${PACKAGE_BEFORE_PN} ${PN}"
PACKAGES_DYNAMIC = "^${PN}-locale-.*"
FILES = ""

FILES_${PN} = "${bindir}/* ${sbindir}/* ${libexecdir}/* ${libdir}/lib*${SOLIBS} \
${sysconfdir} ${sharedstatedir} ${localstatedir} \
${base_bindir}/* ${base_sbindir}/* \
${base_libdir}/*${SOLIBS} \
${base_prefix}/lib/udev/rules.d ${prefix}/lib/udev/rules.d \
${datadir}/${BPN} ${libdir}/${BPN}/* \
${datadir}/pixmaps ${datadir}/applications \
${datadir}/idl ${datadir}/omf ${datadir}/sounds \
${libdir}/bonobo/servers"

FILES_${PN}-bin = "${bindir}/* ${sbindir}/*"

FILES_${PN}-doc = "${docdir} ${mandir} ${infodir} ${datadir}/gtk-doc \
${datadir}/gnome/help"
SECTION_${PN}-doc = "doc"

FILES_SOLIBSDEV ?= "${base_libdir}/lib*${SOLIBSDEV} ${libdir}/lib*${SOLIBSDEV}"
FILES_${PN}-dev = "${includedir} ${FILES_SOLIBSDEV} ${libdir}/*.la \
${libdir}/*.o ${libdir}/pkgconfig ${datadir}/pkgconfig \
${datadir}/aclocal ${base_libdir}/*.o \
${libdir}/${BPN}/*.la ${base_libdir}/*.la"
SECTION_${PN}-dev = "devel"
ALLOW_EMPTY_${PN}-dev = "1"
RDEPENDS_${PN}-dev = "${PN} (= ${EXTENDPKGV})"

FILES_${PN}-staticdev = "${libdir}/*.a ${base_libdir}/*.a ${libdir}/${BPN}/*.a"
SECTION_${PN}-staticdev = "devel"
RDEPENDS_${PN}-staticdev = "${PN}-dev (= ${EXTENDPKGV})"
```

7.12.2. Combining Multiple Versions of Library Files into One Image

The build system offers the ability to build libraries with different target optimizations or architecture formats and combine these together into one system image. You can link different binaries in the image against the different libraries as needed for specific use cases. This feature is called "Multilib."

An example would be where you have most of a system compiled in 32-bit mode using 32-bit libraries, but you have something large, like a database engine, that needs to be a 64-bit application and uses 64-bit libraries. Multilib allows you to get the best of both 32-bit and 64-bit libraries.

While the Multilib feature is most commonly used for 32 and 64-bit differences, the approach the build system uses facilitates different target optimizations. You could compile some binaries to use one set of libraries and other binaries to use a different set of libraries. The libraries could differ in architecture, compiler options, or other optimizations.

Several examples exist in the `meta-skeleton` layer found in the [Source Directory](#):

- `conf/multilib-example.conf` configuration file
- `conf/multilib-example2.conf` configuration file
- `recipes-multilib/images/core-image-multilib-example.bb` recipe

7.12.2.1. Preparing to Use Multilib

User-specific requirements drive the Multilib feature. Consequently, there is no one "out-of-the-box" configuration that likely exists to meet your needs.

In order to enable Multilib, you first need to ensure your recipe is extended to support multiple libraries. Many standard recipes are already extended and support multiple libraries. You can check in the `meta/conf/multilib.conf` configuration file in the [Source Directory](#) to see how this is done using the `BBCLASSEXTEND` variable. Eventually, all recipes will be covered and this list will not be needed.

For the most part, the Multilib class extension works automatically to extend the package name from `${PN}` to `${MLPREFIX}${PN}`, where `MLPREFIX` is the particular multilib (e.g. "lib32-" or "lib64-"). Standard variables such as `DEPENDS`, `RDEPENDS`, `RPROVIDES`, `RRECOMMENDS`, `PACKAGES`, and `PACKAGES_DYNAMIC` are automatically

extended by the system. If you are extending any manual code in the recipe, you can use the `${MLPREFIX}` variable to ensure those names are extended correctly. This automatic extension code resides in `multilib.bbclass`.

7.12.2.2. Using Multilib¶

After you have set up the recipes, you need to define the actual combination of multiple libraries you want to build. You accomplish this through your `local.conf` configuration file in the [Build Directory](#). An example configuration would be as follows:

```
MACHINE = "qemux86-64"
require conf/multilib.conf
MULTILIBS = "multilib:lib32"
DEFAULTTUNE_virtclass-multilib-lib32 = "x86"
IMAGE_INSTALL_append = " lib32-glib-2.0"
```

This example enables an additional library named `lib32` alongside the normal target packages. When combining these "lib32" alternatives, the example uses "x86" for tuning. For information on this particular tuning, see `meta/conf/machine/include/ia32/arch-ia32.inc`.

The example then includes `lib32-glib-2.0` in all the images, which illustrates one method of including a multiple library dependency. You can use a normal image build to include this dependency, for example:

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

You can also build Multilib packages specifically with a command like this:

```
$ bitbake lib32-glib-2.0
```

7.12.2.3. Additional Implementation Details¶

Generic implementation details as well as details that are specific to package management systems exist. Following are implementation details that exist regardless of the package management system:

- The typical convention used for the class extension code as used by Multilib assumes that all package names specified in [PACKAGES](#) that contain `PN` have `PN` at the start of the name. When that convention is not followed and `PN` appears at the middle or the end of a name, problems occur.
- The [TARGET_VENDOR](#) value under Multilib will be extended to `-"vendormultilib"` (e.g. `-"pokymllib32"` for a "lib32" Multilib with Poky). The reason for this slightly unwieldy contraction is that any `-"` characters in the vendor string presently break Autoconf's `config.sub`, and other separators are problematic for different reasons.

For the RPM Package Management System, the following implementation details exist:

- A unique architecture is defined for the Multilib packages, along with creating a unique deploy folder under `tmp/deploy/rpm` in the [Build Directory](#). For example, consider `lib32` in a `qemux86-64` image. The possible architectures in the system are "all", `qemux86_64`, `lib32_qemux86_64`, and `lib32_x86`.
- The `MLPREFIX` variable is stripped from `PN` during RPM packaging. The naming for a normal RPM package and a Multilib RPM package in a `qemux86-64` system resolves to something similar to `bash-4.1-r2.x86_64.rpm` and `bash-4.1.r2.lib32_x86.rpm`, respectively.
- When installing a Multilib image, the RPM backend first installs the base image and then installs the Multilib libraries.
- The build system relies on RPM to resolve the identical files in the two (or more) Multilib packages.

For the IPK Package Management System, the following implementation details exist:

- The `MLPREFIX` is not stripped from `PN` during IPK packaging. The naming for a normal RPM package and a Multilib IPK package in a `qemux86-64` system resolves to something like `bash_4.1-r2.x86_64.ipk` and `lib32-bash_4.1-rw_x86.ipk`, respectively.
- The IPK deploy folder is not modified with `MLPREFIX` because packages with and without the Multilib feature can exist in the same folder due to the `PN` differences.
- IPK defines a sanity check for Multilib installation using certain rules for file comparison, overridden, etc.

7.12.3. Installing Multiple Versions of the Same Library¶

Situations can exist where you need to install and use multiple versions of the same library on the same system at the same time. These situations almost always exist when a library API changes and you have multiple pieces of software that depend on the separate versions of the library. To accommodate these situations, you can install multiple versions of the same library in parallel on the same system.

The process is straightforward as long as the libraries use proper versioning. With properly versioned libraries, all you need to do to individually specify the libraries is create separate, appropriately named recipes where the `PN` part of the name includes a portion that differentiates each library version (e.g. the major part of the version number). Thus, instead of having a single recipe that loads one version of a library (e.g. `clutter`), you provide multiple recipes that result in different versions of the libraries you want. As an example, the following two recipes would allow the two separate versions of the `clutter` library to co-exist on the same system:

```
clutter-1.6_1.6.20.bb
clutter-1.8_1.8.4.bb
```

Additionally, if you have other recipes that depend on a given library, you need to use the [DEPENDS](#) variable to create the dependency. Continuing with the same example, if you want to have a recipe depend on the 1.8 version of the `clutter` library, use the following in your recipe:

```
DEPENDS = "clutter-1.8"
```

7.13. Using x32 psABI¶

x32 processor-specific Application Binary Interface ([x32 psABI](#)) is a native 32-bit processor-specific ABI for Intel® 64 (x86-64) architectures. An ABI defines the calling conventions between functions in a processing environment. The interface determines what registers are used and what the sizes are for various C data types.

Some processing environments prefer using 32-bit applications even when running on Intel 64-bit platforms. Consider the i386 psABI, which is a very old 32-bit ABI for Intel 64-bit platforms. The i386 psABI does not provide efficient use and access of the Intel 64-bit processor resources, leaving the system underutilized. Now consider the x86_64 psABI. This ABI is newer and uses 64-bits for data sizes and program pointers. The extra bits increase the footprint size of the programs, libraries, and also increases the memory and file system size requirements. Executing under the x32 psABI enables user programs to utilize CPU and system resources more efficiently while keeping the memory footprint of the applications low. Extra bits are used for registers but not for addressing mechanisms.

The Yocto Project supports the final specifications of x32 psABI as follows:

- You can create packages and images in x32 psABI format on x86_64 architecture targets.
- You can successfully build recipes with the x32 toolchain.
- You can create and boot `core-image-minimal` and `core-image-sato` images.
- RPM Package Manager (RPM) support exists for x32 binaries.
- Support for large images exists.

To use the x32 psABI, you need to edit your `conf/local.conf` configuration file as follows:

```
MACHINE = "qemux86-64"
DEFAULTTUNE = "x86-64-x32"
baselib = "${@d.getVar('BASE_LIB_tune-' + (d.getVar('DEFAULTTUNE') \
    or 'INVALID')) or 'lib'}
```

Once you have set up your configuration file, use BitBake to build an image that supports the x32 psABI. Here is an example:

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

7.14. Enabling GObject Introspection Support¶

[GObject introspection](#) is the standard mechanism for accessing GObject-based software from runtime environments. GObject is a feature of the GLib library that provides an object framework for the GNOME desktop and related software. GObject Introspection adds information to GObject that allows objects created within it to be represented across different programming languages. If you want to construct GStreamer pipelines using Python, or control UPnP infrastructure using Javascript and GUPnP, GObject introspection is the only way to do it.

This section describes the Yocto Project support for generating and packaging GObject introspection data. GObject introspection data is a description of the API provided by libraries built on top of GLib framework, and, in particular, that framework's GObject mechanism. GObject Introspection Repository (GIR) files go to `-dev` packages, `typelib` files go to main packages as they are packaged together with libraries that are introspected.

The data is generated when building such a library, by linking the library with a small executable binary that asks the library to describe itself, and then executing the binary and processing its output.

Generating this data in a cross-compilation environment is difficult because the library is produced for the target architecture, but its code needs to be executed on the build host. This problem is solved with the OpenEmbedded build system by running the code through QEMU, which allows precisely that. Unfortunately, QEMU does not always work perfectly as mentioned in the ["Known Issues"](#) section.

7.14.1. Enabling the Generation of Introspection Data¶

Enabling the generation of introspection data (GIR files) in your library package involves the following:

1. Inherit the `gobject-introspection` class.
2. Make sure introspection is not disabled anywhere in the recipe or from anything the recipe includes. Also, make sure that "gobject-introspection-data" is not in `DISTRO_FEATURES_BACKFILL_CONSIDERED` and that "qemu-usermode" is not in `MACHINE_FEATURES_BACKFILL_CONSIDERED`. If either of these conditions exist, nothing will happen.
3. Try to build the recipe. If you encounter build errors that look like something is unable to find `.so` libraries, check where these libraries are located in the source tree and add the following to the recipe:

```
GIR_EXTRA_LIBS_PATH = "${B}/something/.libs"
```

Note

See recipes in the `oe-core` repository that use that `GIR_EXTRA_LIBS_PATH` variable as an example.

4. Look for any other errors, which probably mean that introspection support in a package is not entirely standard, and thus breaks down in a cross-compilation environment. For such cases, custom-made fixes are needed. A good place to ask and receive help in these cases is the [Yocto Project mailing lists](#).

Note

Using a library that no longer builds against the latest Yocto Project release and prints introspection related errors is a good candidate for the previous procedure.

7.14.2. Disabling the Generation of Introspection Data¶

You might find that you do not want to generate introspection data. Or, perhaps QEMU does not work on your build host and target architecture combination. If so, you can use either of the following methods to disable GIR file generations:

- Add the following to your distro configuration:

```
DISTRO_FEATURES_BACKFILL_CONSIDERED = "gobject-introspection-data"
```

Adding this statement disables generating introspection data using QEMU but will still enable building introspection tools and libraries (i.e. building them does not require the use of QEMU).

- Add the following to your machine configuration:

```
MACHINE_FEATURES_BACKFILL_CONSIDERED = "qemu-usermode"
```

Adding this statement disables the use of QEMU when building packages for your machine. Currently, this feature is used only by introspection recipes and has the same effect as the previously described option.

Note

Future releases of the Yocto Project might have other features affected by this option.

If you disable introspection data, you can still obtain it through other means such as copying the data from a suitable sysroot, or by generating it on the target hardware. The OpenEmbedded build system does not currently provide specific support for these techniques.

7.14.3. Testing that Introspection Works in an Image¶

Use the following procedure to test if generating introspection data is working in an image:

1. Make sure that "gobject-introspection-data" is not in `DISTRO_FEATURES_BACKFILL_CONSIDERED` and that "qemu-usermode" is not in `MACHINE_FEATURES_BACKFILL_CONSIDERED`.
2. Build `core-image-sato`.
3. Launch a Terminal and then start Python in the terminal.
4. Enter the following in the terminal:

```
>>> from gi.repository import GLib
>>> GLib.get_host_name()
```

5. For something a little more advanced, enter the following:

```
http://python-gtk-3-tutorial.readthedocs.org/en/latest/introduction.html
```

7.14.4. Known Issues¶

The following known issues exist for GObject Introspection Support:

- `qemu-ppc64` immediately crashes. Consequently, you cannot build introspection data on that architecture.
- `x32` is not supported by QEMU. Consequently, introspection data is disabled.
- `musl` causes transient GLib binaries to crash on assertion failures. Consequently, generating introspection data is disabled.
- Because QEMU is not able to run the binaries correctly, introspection is disabled for some specific packages under specific architectures (e.g. `gcr`, `libsecret`, and `webkit`).
- QEMU usermode might not work properly when running 64-bit binaries under 32-bit host machines. In particular, "qemumips64" is known to not work under i686.

7.15. Optionally Using an External Toolchain¶

You might want to use an external toolchain as part of your development. If this is the case, the fundamental steps you need to accomplish are as follows:

- Understand where the installed toolchain resides. For cases where you need to build the external toolchain, you would need to take separate steps to build and install the toolchain.
- Make sure you add the layer that contains the toolchain to your `bblayers.conf` file through the `BBLAYERS` variable.
- Set the `EXTERNAL_TOOLCHAIN` variable in your `local.conf` file to the location in which you installed the toolchain.

A good example of an external toolchain used with the Yocto Project is Mentor Graphics® Sourcery G++ Toolchain. You can see information on how to use that particular layer in the README file at <http://github.com/MentorEmbedded/meta-sourcery/>. You can find further information by reading about the `TCMODE` variable in the Yocto Project Reference Manual's variable glossary.

7.16. Creating Partitioned Images Using Wic¶

Creating an image for a particular hardware target using the OpenEmbedded build system does not necessarily mean you can boot that image as is on your device. Physical devices accept and boot images in various ways depending on the specifics of the device. Usually,

information about the hardware can tell you what image format the device requires. Should your device require multiple partitions on an SD card, flash, or an HDD, you can use the OpenEmbedded Image Creator, Wic, to create the properly partitioned image.

The `wic` command generates partitioned images from existing OpenEmbedded build artifacts. Image generation is driven by partitioning commands contained in an OpenEmbedded kickstart file (`.wks`) specified either directly on the command line or as one of a selection of canned kickstart files as shown with the `wic list images` command in the "[Using an Existing Kickstart File](#)" section. When you apply the command to a given set of build artifacts, the result is an image or set of images that can be directly written onto media and used on a particular system.

Note

For a kickstart file reference, see the "[OpenEmbedded Kickstart \(.wks\) Reference](#)" Chapter in the Yocto Project Reference Manual.

The `wic` command and the infrastructure it is based on is by definition incomplete. The purpose of the command is to allow the generation of customized images, and as such, was designed to be completely extensible through a plugin interface. See the "[Using the Wic PlugIn Interface](#)" section for information on these plugins.

This section provides some background information on Wic, describes what you need to have in place to run the tool, provides instruction on how to use the Wic utility, provides information on using the Wic plugins interface, and provides several examples that show how to use Wic.

7.16.1. Background¶

This section provides some background on the Wic utility. While none of this information is required to use Wic, you might find it interesting.

- The name "Wic" is derived from OpenEmbedded Image Creator (oeic). The "oe" diphthong in "oeic" was promoted to the letter "w", because "oeic" is both difficult to remember and to pronounce.
- Wic is loosely based on the Meego Image Creator (`mic`) framework. The Wic implementation has been heavily modified to make direct use of OpenEmbedded build artifacts instead of package installation and configuration, which are already incorporated within the OpenEmbedded artifacts.
- Wic is a completely independent standalone utility that initially provides easier-to-use and more flexible replacements for an existing functionality in OE-Core's `image-live` class. The difference between Wic and those examples is that with Wic the functionality of those scripts is implemented by a general-purpose partitioning language, which is based on Redhat kickstart syntax.

7.16.2. Requirements¶

In order to use the Wic utility with the OpenEmbedded Build system, your system needs to meet the following requirements:

- The Linux distribution on your development host must support the Yocto Project. See the "[Supported Linux Distributions](#)" section in the Yocto Project Reference Manual for the list of distributions that support the Yocto Project.
- The standard system utilities, such as `cp`, must be installed on your development host system.
- You must have sourced the build environment setup script (i.e. `oe-init-build-env`) found in the [Build Directory](#).
- You need to have the build artifacts already available, which typically means that you must have already created an image using the OpenEmbedded build system (e.g. `core-image-minimal`). While it might seem redundant to generate an image in order to create an image using Wic, the current version of Wic requires the artifacts in the form generated by the OpenEmbedded build system.
- You must build several native tools, which are built to run on the build system:

```
$ bitbake parted-native dosfstools-native mtools-native
```

- Include "wic" as part of the `IMAGE_FSTYPES` variable.
- Include the name of the [wic kickstart file](#) as part of the `WKS_FILE` variable

7.16.3. Getting Help¶

You can get general help for the `wic` command by entering the `wic` command by itself or by entering the command with a help argument as follows:

```
$ wic -h
$ wic --help
$ wic help
```

Currently, Wic supports seven commands: `cp`, `create`, `help`, `list`, `ls`, `rm`, and `write`. You can get help for all these commands except "help" by using the following form:

```
$ wic help command
```

For example, the following command returns help for the `write` command:

```
$ wic help write
```

Wic supports help for three topics: `overview`, `plugins`, and `kickstart`. You can get help for any topic using the following form:

```
$ wic help topic
```

For example, the following returns overview help for Wic:

```
$ wic help overview
```

One additional level of help exists for Wic. You can get help on individual images through the `list` command. You can use the `list` command to return the available Wic images as follows:

```
$ wic list images
mpc8315e-rdb          Create SD card image for MPC8315E-RDB
genericx86            Create an EFI disk image for genericx86*
beaglebone-yocto     Create SD card image for Beaglebone
edgerouter           Create SD card image for Edgerouter
qemu86-directdisk    Create a qemu machine 'pcbios' direct disk image
directdisk-gpt       Create a 'pcbios' direct disk image
mkefidisk            Create an EFI disk image
directdisk           Create a 'pcbios' direct disk image
systemd-bootdisk     Create an EFI disk image with systemd-boot
mkhybridiso          Create a hybrid ISO image
sdimage-bootpart     Create SD card image with a boot partition
directdisk-multi-rootfs Create multi rootfs image using rootfs plugin
directdisk-bootloader-config Create a 'pcbios' direct disk image with custom bootloader config
```

Once you know the list of available Wic images, you can use `help` with the command to get help on a particular image. For example, the following command returns help on the "beaglebone-yocto" image:

```
$ wic list beaglebone-yocto help
```

```
Creates a partitioned SD card image for Beaglebone.
Boot files are located in the first vfat partition.
```

7.16.4. Operational Modes¶

You can use Wic in two different modes, depending on how much control you need for specifying the Openembedded build artifacts that are used for creating the image: Raw and Cooked:

- **Raw Mode:** You explicitly specify build artifacts through Wic command-line arguments.
- **Cooked Mode:** The current `MACHINE` setting and image name are used to automatically locate and provide the build artifacts. You just supply a kickstart file and the name of the image from which to use artifacts.

Regardless of the mode you use, you need to have the build artifacts ready and available.

7.16.4.1. Raw Mode¶

Running Wic in raw mode allows you to specify all the partitions through the `wic` command line. The primary use for raw mode is if you have built your kernel outside of the Yocto Project [Build Directory](#). In other words, you can point to arbitrary kernel, root filesystem locations, and so forth. Contrast this behavior with cooked mode where Wic looks in the Build Directory (e.g. `tmp/deploy/images/machine`).

The general form of the `wic` command in raw mode is:

```
$ wic create wks_file options ...
```

Where:

wks_file:

An OpenEmbedded kickstart file. You can provide your own custom file or use a file from a set of existing files as described by further options.

optional arguments:

```
-h, --help          show this help message and exit
-o OUTDIR, --outdir OUTDIR
                    name of directory to create image in
-e IMAGE_NAME, --image-name IMAGE_NAME
                    name of the image to use the artifacts from e.g. core-image-sato
-r ROOTFS_DIR, --rootfs-dir ROOTFS_DIR
                    path to the /rootfs dir to use as the .wks rootfs source
-b BOOTIMG_DIR, --bootimg-dir BOOTIMG_DIR
                    path to the dir containing the boot artifacts (e.g. /EFI or /syslinux dirs) to use as the .wks bootimg source
-k KERNEL_DIR, --kernel-dir KERNEL_DIR
                    path to the dir containing the kernel to use in the .wks bootimg
-n NATIVE_SYSROOT, --native-sysroot NATIVE_SYSROOT
                    path to the native sysroot containing the tools to use to build the image
-s, --skip-build-check
                    skip the build check
-f, --build-rootfs  build rootfs
-c {gzip,bzip2,xz}, --compress-with {gzip,bzip2,xz}
                    compress image with specified compressor
-m, --bmap          generate .bmap
--no-fstab-update   Do not change fstab file.
-v VARS_DIR, --vars VARS_DIR
                    directory with <image>.env files that store bitbake variables
-D, --debug         output debug information
```


Note

You do not need root privileges to run Wic. In fact, you should not run as root when using the utility.

7.16.4.2. Cooked Mode¶

Running Wic in cooked mode leverages off artifacts in the Build Directory. In other words, you do not have to specify kernel or root filesystem locations as part of the command. All you need to provide is a kickstart file and the name of the image from which to use artifacts by using the "-e" option. Wic looks in the Build Directory (e.g. `tmp/deploy/images/machine`) for artifacts.

The general form of the `wic` command using Cooked Mode is as follows:

```
$ wic create wks_file -e IMAGE_NAME
```

Where:

`wks_file`:

An OpenEmbedded kickstart file. You can provide your own custom file or use a file from a set of existing files provided with the Yocto Project release.

required argument:

```
-e IMAGE_NAME, --image-name IMAGE_NAME
                        name of the image to use the artifacts from e.g. core-
                        image-sato
```

7.16.5. Using an Existing Kickstart File¶

If you do not want to create your own kickstart file, you can use an existing file provided by the Wic installation. As shipped, kickstart files can be found in the Yocto Project [Source Repositories](#) in the following two locations:

```
poky/meta-yocto-bsp/wic
poky/scripts/lib/wic/canned-wks
```

Use the following command to list the available kickstart files:

```
$ wic list images
mpc8315e-rdb          Create SD card image for MPC8315E-RDB
genericx86           Create an EFI disk image for genericx86*
beaglebone-yocto     Create SD card image for Beaglebone
edgerouter           Create SD card image for Edgerouter
qemux86-directdisk   Create a qemu machine 'pcbios' direct disk image
directdisk-gpt       Create a 'pcbios' direct disk image
mkefidisk           Create an EFI disk image
directdisk           Create a 'pcbios' direct disk image
systemd-bootdisk     Create an EFI disk image with systemd-boot
mkhybridiso          Create a hybrid ISO image
sdimage-bootpart     Create SD card image with a boot partition
directdisk-multi-rootfs Create multi rootfs image using rootfs plugin
directdisk-bootloader-config Create a 'pcbios' direct disk image with custom bootloader config
```

When you use an existing file, you do not have to use the `.wks` extension. Here is an example in Raw Mode that uses the `directdisk` file:

```
$ wic create directdisk -r rootfs_dir -b bootimg_dir \
-k kernel_dir -n native_sysroot
```

Here are the actual partition language commands used in the `genericx86.wks` file to generate an image:

```
# short-description: Create an EFI disk image for genericx86*
# long-description: Creates a partitioned EFI disk image for genericx86* machines
part /boot --source bootimg-efi --sourceparams="loader=grub-efi" --ondisk sda --label msdos --active --align 1024
part / --source rootfs --ondisk sda --fstype=ext4 --label platform --align 1024 --use-uuid
part swap --ondisk sda --size 44 --label swap1 --fstype=swap

bootloader --ptable gpt --timeout=5 --append="rootfstype=ext4 console=ttyS0,115200 console=tty0"
```

7.16.6. Using the Wic Plugin Interface¶

You can extend and specialize Wic functionality by using Wic plugins. This section explains the Wic plugin interface.

Note

Wic plugins consist of "source" and "imager" plugins. Imager plugins are beyond the scope of this section.

Source plugins provide a mechanism to customize partition content during the Wic image generation process. You can use source plugins to map values that you specify using `--SOURCE` commands in kickstart files (i.e. `*.wks`) to a plugin implementation used to populate a given partition.

Note

If you use plugins that have build-time dependencies (e.g. native tools, bootloaders, and so forth) when building a Wic image, you need to specify those dependencies using the `WKS_FILE_DEPENDS` variable.

Source plugins are subclasses defined in plugin files. As shipped, the Yocto Project provides several plugin files. You can see the source plugin files that ship with the Yocto Project [here](#). Each of these plugin files contains source plugins that are designed to populate a specific Wic image partition.

Source plugins are subclasses of the `SourcePlugin` class, which is defined in the `poky/scripts/lib/wic/pluginbase.py` file. For example, the `BootimgEFIPlugin` source plugin found in the `bootimg-efi.py` file is a subclass of the `SourcePlugin` class, which is found in the `pluginbase.py` file.

You can also implement source plugins in a layer outside of the Source Repositories (external layer). To do so, be sure that your plugin files are located in a directory whose path is `scripts/lib/wic/plugins/source/` within your external layer. When the plugin files are located there, the source plugins they contain are made available to Wic.

When the Wic implementation needs to invoke a partition-specific implementation, it looks for the plugin with the same name as the `--source` parameter used in the kickstart file given to that partition. For example, if the partition is set up using the following command in a kickstart file:

```
part /boot --source bootimg-pcbios --ondisk sda --label boot --active --align 1024
```

The methods defined as class members of the matching source plugin (i.e. `bootimg-pcbios`) in the `bootimg-pcbios.py` plugin file are used.

To be more concrete, here is the corresponding plugin definition from the `bootimg-pcbios.py` file for the previous command along with an example method called by the Wic implementation when it needs to prepare a partition using an implementation-specific function:

```

    .
    .
    .
class BootimgPcbiosPlugin(SourcePlugin):
    """
    Create MBR boot partition and install syslinux on it.
    """

    name = 'bootimg-pcbios'
    .
    .
    .
    @classmethod
    def do_prepare_partition(cls, part, source_params, creator, cr_workdir,
                           oe_builddir, bootimg_dir, kernel_dir,
                           rootfs_dir, native_sysroot):
        """
        Called to do the actual content population for a partition i.e. it
        'prepares' the partition to be incorporated into the image.
        In this case, prepare content for legacy bios boot partition.
        """
    .
    .
    .

```

If a subclass (plugin) itself does not implement a particular function, Wic locates and uses the default version in the superclass. It is for this reason that all source plugins are derived from the `SourcePlugin` class.

The `SourcePlugin` class defined in the `pluginbase.py` file defines a set of methods that source plugins can implement or override. Any plugins (subclass of `SourcePlugin`) that do not implement a particular method inherit the implementation of the method from the `SourcePlugin` class. For more information, see the `SourcePlugin` class in the `pluginbase.py` file for details:

The following list describes the methods implemented in the `SourcePlugin` class:

- **`do_prepare_partition()`**: Called to populate a partition with actual content. In other words, the method prepares the final partition image that is incorporated into the disk image.
- **`do_configure_partition()`**: Called before `do_prepare_partition()` to create custom configuration files for a partition (e.g. syslinux or grub configuration files).
- **`do_install_disk()`**: Called after all partitions have been prepared and assembled into a disk image. This method provides a hook to allow finalization of a disk image (e.g. writing an MBR).
- **`do_stage_partition()`**: Special content-staging hook called before `do_prepare_partition()`. This method is normally empty.

Typically, a partition just uses the passed-in parameters (e.g. the unmodified value of `bootimg_dir`). However, in some cases, things might need to be more tailored. As an example, certain files might additionally need to be taken from `bootimg_dir + /boot`. This hook allows those files to be staged in a customized fashion.

Note

`get_bitbake_var()` allows you to access non-standard variables that you might want to use for this behavior.

You can extend the source plugin mechanism. To add more hooks, create more source plugin methods within `SourcePlugin` and the corresponding derived subclasses. The code that calls the plugin methods uses the `plugin.get_source_plugin_methods()` function to find the method or methods needed by the call. Retrieval of those methods is accomplished by filling up a dict with keys that contain the method names of interest. On success, these will be filled in with the actual methods. See the Wic implementation for examples and details.

7.16.7. Examples¶

This section provides several examples that show how to use the Wic utility. All the examples assume the list of requirements in the "Requirements" section have been met. The examples assume the previously generated image is `core-image-minimal`.

7.16.7.1. Generate an Image using an Existing Kickstart File¶

This example runs in Cooked Mode and uses the `mkefidisk` kickstart file:

```
$ wic create mkefidisk -e core-image-minimal
INFO: Building wic-tools...
.
.
.
INFO: The new image(s) can be found here:
./mkefidisk-201804191017-sda.direct

The following build artifacts were used to create the image(s):
ROOTFS_DIR: /home/stephano/build/master/build/tmp-glibc/work/qemux86-oe-linux/core-image-minimal/1.0-1
BOOTIMG_DIR: /home/stephano/build/master/build/tmp-glibc/work/qemux86-oe-linux/core-image-minimal/1.0-1
KERNEL_DIR: /home/stephano/build/master/build/tmp-glibc/deploy/images/qemux86
NATIVE_SYSDIR: /home/stephano/build/master/build/tmp-glibc/work/i586-oe-linux/wic-tools/1.0-r0/recipe-sy

INFO: The image(s) were created using OE kickstart file:
/home/stephano/build/master/openembedded-core/scripts/lib/wic/canned-wks/mkefidisk.wks
```

The previous example shows the easiest way to create an image by running in cooked mode and supplying a kickstart file and the "-e" option to point to the existing build artifacts. Your `local.conf` file needs to have the `MACHINE` variable set to the machine you are using, which is "qemux86" in this example.

Once the image builds, the output provides image location, artifact use, and kickstart file information.

Note

You should always verify the details provided in the output to make sure that the image was indeed created exactly as expected.

Continuing with the example, you can now write the image from the Build Directory onto a USB stick, or whatever media for which you built your image, and boot from the media. You can write the image by using `bmptool` or `dd`:

```
$ oe-run-native bmptool copy mkefidisk-201804191017-sda.direct /dev/sdX
```

or

```
$ sudo dd if=mkefidisk-201804191017-sda.direct of=/dev/sdX
```

Note

For more information on how to use the `bmptool` to flash a device with an image, see the "Flashing Images Using `bmptool`" section.

7.16.7.2. Using a Modified Kickstart File¶

Because partitioned image creation is driven by the kickstart file, it is easy to affect image creation by changing the parameters in the file. This next example demonstrates that through modification of the `directdisk-gpt` kickstart file.

As mentioned earlier, you can use the command `wic list images` to show the list of existing kickstart files. The directory in which the `directdisk-gpt.wks` file resides is `scripts/lib/image/canned-wks/`, which is located in the [Source Directory](#) (e.g. `poky`). Because available files reside in this directory, you can create and add your own custom files to the directory. Subsequent use of the `wic list images` command would then include your kickstart files.

In this example, the existing `directdisk-gpt` file already does most of what is needed. However, for the hardware in this example, the image will need to boot from `sdb` instead of `sda`, which is what the `directdisk-gpt` kickstart file uses.

The example begins by making a copy of the `directdisk-gpt.wks` file in the `scripts/lib/image/canned-wks` directory and then by changing the lines that specify the target disk from which to boot.

```
$ cp /home/stephano/poky/scripts/lib/wic/canned-wks/directdisk-gpt.wks \
/home/stephano/poky/scripts/lib/wic/canned-wks/directdisksdb-gpt.wks
```

Next, the example modifies the `directdisksdb-gpt.wks` file and changes all instances of "`--ondisk sda`" to "`--ondisk sdb`". The example changes the following two lines and leaves the remaining lines untouched:

```
part /boot --source bootimg-pcbios --ondisk sdb --label boot --active --align 1024
part / --source rootfs --ondisk sdb --fstype=ext4 --label platform --align 1024 --use-uuid
```

Once the lines are changed, the example generates the `directdisksdb-gpt` image. The command points the process at the `core-image-minimal` artifacts for the Next Unit of Computing (nuc) `MACHINE` the `local.conf`.

```
$ wic create directdisksdb-gpt -e core-image-minimal
INFO: Building wic-tools...
.
.
.
```

```

Initialising tasks: 100% |#####| Time: 0:00:01
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
NOTE: Tasks Summary: Attempted 1161 tasks of which 1157 didn't need to be rerun and all succeeded.
INFO: Creating image(s)...

INFO: The new image(s) can be found here:
./directdisksdb-gpt-201710090938-sdb.direct

The following build artifacts were used to create the image(s):
ROOTFS_DIR: /home/stephano/build/master/build/tmp-glibc/work/qemux86-oe-linux/core-image-minimal/1.0-r0
BOOTIMG_DIR: /home/stephano/build/master/build/tmp-glibc/work/qemux86-oe-linux/core-image-minimal/1.0-r0
KERNEL_DIR: /home/stephano/build/master/build/tmp-glibc/deploy/images/qemux86
NATIVE_SYSDIR: /home/stephano/build/master/build/tmp-glibc/work/i586-oe-linux/wic-tools/1.0-r0/recipe-sysroot

INFO: The image(s) were created using OE kickstart file:
/home/stephano/poky/scripts/lib/wic/canned-wks/directdisksdb-gpt.wks

```

Continuing with the example, you can now directly `dd` the image to a USB stick, or whatever media for which you built your image, and boot the resulting media:

```

$ sudo dd if=directdisksdb-gpt-201710090938-sdb.direct of=/dev/sdb
140966+0 records in
140966+0 records out
72174592 bytes (72 MB, 69 MiB) copied, 78.0282 s, 925 kB/s
$ sudo eject /dev/sdb

```

7.16.7.3. Using a Modified Kickstart File and Running in Raw Mode¶

This next example manually specifies each build artifact (runs in Raw Mode) and uses a modified kickstart file. The example also uses the `-O` option to cause Wic to create the output somewhere other than the default output directory, which is the current directory:

```

$ wic create /home/stephano/my_yocto/test.wks -o /home/stephano/testwic \
--rootfs-dir /home/stephano/build/master/build/tmp/work/qemux86-poky-linux/core-image-minimal/1.0-r0/rootfs \
--bootimg-dir /home/stephano/build/master/build/tmp/work/qemux86-poky-linux/core-image-minimal/1.0-r0/recipe-sysroot \
--kernel-dir /home/stephano/build/master/build/tmp/deploy/images/qemux86 \
--native-sysroot /home/stephano/build/master/build/tmp/work/i586-poky-linux/wic-tools/1.0-r0/recipe-sysroot-native

INFO: Creating image(s)...

INFO: The new image(s) can be found here:
/home/stephano/testwic/test-201710091445-sdb.direct

The following build artifacts were used to create the image(s):
ROOTFS_DIR: /home/stephano/build/master/build/tmp-glibc/work/qemux86-oe-linux/core-image-minimal/1.0-r0
BOOTIMG_DIR: /home/stephano/build/master/build/tmp-glibc/work/qemux86-oe-linux/core-image-minimal/1.0-r0
KERNEL_DIR: /home/stephano/build/master/build/tmp-glibc/deploy/images/qemux86
NATIVE_SYSDIR: /home/stephano/build/master/build/tmp-glibc/work/i586-oe-linux/wic-tools/1.0-r0/recipe-sysroot

INFO: The image(s) were created using OE kickstart file:
/home/stephano/my_yocto/test.wks

```

For this example, `MACHINE` did not have to be specified in the `local.conf` file since the artifact is manually specified.

7.16.7.4. Using Wic to Manipulate an Image¶

Wic image manipulation allows you to shorten turnaround time during image development. For example, you can use Wic to delete the kernel partition of a Wic image and then insert a newly built kernel. This saves you time from having to rebuild the entire image each time you modify the kernel.

Note

In order to use Wic to manipulate a Wic image as in this example, your development machine must have the `mttools` package installed.

The following example examines the contents of the Wic image, deletes the existing kernel, and then inserts a new kernel:

1. **List the Partitions:** Use the `wic ls` command to list all the partitions in the Wic image:

```

$ wic ls tmp/deploy/images/qemux86/core-image-minimal-qemux86.wic
Num   Start      End      Size      Fstype
1     1048576    25041919 23993344  fat16
2     25165824    72157183 46991360  ext4

```

The previous output shows two partitions in the `core-image-minimal-qemux86.wic` image.

2. **Examine a Particular Partition:** Use the `wic ls` command again but in a different form to examine a particular partition.

Note

You can get command usage on any Wic command using the following form:

```
$ wic help command
```

For example, the following command shows you the various ways to use the `wic ls` command:

```
$ wic help ls
```

The following command shows what is in Partition one:

```
$ wic ls tmp/deploy/images/qemux86/core-image-minimal-qemux86.wic:1
Volume in drive : is boot
Volume Serial Number is E894-1809
Directory for ::/

libcom32 c32      186500 2017-10-09 16:06
libutil c32       24148 2017-10-09 16:06
syslinux cfg      220 2017-10-09 16:06
vesamenu c32      27104 2017-10-09 16:06
vmlinuz          6904608 2017-10-09 16:06
5 files              7 142 580 bytes
16 582 656 bytes free
```

The previous output shows five files, with the `vmlinuz` being the kernel.

Note

If you see the following error, you need to update or create a `~/.mtoolsrc` file and be sure to have the line `"mtools_skip_check=1"` in the file. Then, run the Wic command again:

```
ERROR: _exec_cmd: /usr/bin/mdir -i /tmp/wic-parttfokuwra ::/ returned '1' instead of 0
output: Total number of sectors (47824) not a multiple of sectors per track (32)!
Add mtools_skip_check=1 to your .mtoolsrc file to skip this test
```

3. **Remove the Old Kernel:** Use the `wic rm` command to remove the `vmlinuz` file (kernel):

```
$ wic rm tmp/deploy/images/qemux86/core-image-minimal-qemux86.wic:1/vmlinuz
```

4. **Add In the New Kernel:** Use the `wic cp` command to add the updated kernel to the Wic image. Depending on how you built your kernel, it could be in different places. If you used `devtool` and an SDK to build your kernel, it resides in the `tmp/work` directory of the extensible SDK. If you used `make` to build the kernel, the kernel will be in the `workspace/sources` area.

The following example assumes `devtool` was used to build the kernel:

```
cp ~/poky_sdk/tmp/work/qemux86-poky-linux/linux-yocto/4.12.12+git999-r0/linux-yocto-4.12.12+git999/arch/x86/boot/b:
~/poky/build/tmp/deploy/images/qemux86/core-image-minimal-qemux86.wic:1/vmlinuz
```

Once the new kernel is added back into the image, you can use the `dd` command or [bmaptool](#) to flash your wic image onto an SD card or USB stick and test your target.

Note

Using `bmaptool` is generally 10 to 20 times faster than using `dd`.

7.17. Flashing Images Using `bmaptool`

A fast and easy way to flash an image to a bootable device is to use Bmaptool, which is integrated into the OpenEmbedded build system. Bmaptool is a generic tool that creates a file's block map (bmap) and then uses that map to copy the file. As compared to traditional tools such as `dd` or `cp`, Bmaptool can copy (or flash) large files like raw system image files much faster.

Notes

- If you are using Ubuntu or Debian distributions, you can install the `bmap-tools` package using the following command and then use the tool without specifying `PATH` even from the root account:

```
$ sudo apt-get install bmap-tools
```

- If you are unable to install the `bmap-tools` package, you will need to build Bmaptool before using it. Use the following command:

```
$ bitbake bmap-tools-native
```

Following, is an example that shows how to flash a Wic image. Realize that while this example uses a Wic image, you can use Bmaptool to flash any type of image. Use these steps to flash an image using Bmaptool:

1. **Update your local .conf File:** You need to have the following set in your `local.conf` file before building your image:

```
IMAGE_FSTYPES += "wic wic.bmap"
```

2. **Get Your Image:** Either have your image ready (pre-built with the `IMAGE_FSTYPES` setting previously mentioned) or take the step to build the image:

```
$ bitbake image
```

3. **Flash the Device:** Flash the device with the image by using Bmaptool depending on your particular setup. The following commands assume the image resides in the Build Directory's `deploy/images/` area:

- If you have write access to the media, use this command form:

```
$ oe-run-native bmap-tools-native bmaptool copy build-directory/tmp/deploy/images/machine/image.wic /dev/sdX
```

- If you do not have write access to the media, set your permissions first and then use the same command form:

```
$ sudo chmod 666 /dev/sdX
$ oe-run-native bmap-tools-native bmaptool copy build-directory/tmp/deploy/images/machine/image.wic /dev/sdX
```

For help on the `bmaptool` command, use the following command:

```
$ bmaptool --help
```

7.18. Making Images More Secure¶

Security is of increasing concern for embedded devices. Consider the issues and problems discussed in just this sampling of work found across the Internet:

- **"Security Risks of Embedded Systems"** by Bruce Schneier
- **"Internet Census 2012"** by Carna Botnet
- **"Security Issues for Embedded Devices"** by Jake Edge

When securing your image is of concern, there are steps, tools, and variables that you can consider to help you reach the security goals you need for your particular device. Not all situations are identical when it comes to making an image secure. Consequently, this section provides some guidance and suggestions for consideration when you want to make your image more secure.

Note

Because the security requirements and risks are different for every type of device, this section cannot provide a complete reference on securing your custom OS. It is strongly recommended that you also consult other sources of information on embedded Linux system hardening and on security.

7.18.1. General Considerations¶

General considerations exist that help you create more secure images. You should consider the following suggestions to help make your device more secure:

- Scan additional code you are adding to the system (e.g. application code) by using static analysis tools. Look for buffer overflows and other potential security problems.
- Pay particular attention to the security for any web-based administration interface.

Web interfaces typically need to perform administrative functions and tend to need to run with elevated privileges. Thus, the consequences resulting from the interface's security becoming compromised can be serious. Look for common web vulnerabilities such as cross-site-scripting (XSS), unvalidated inputs, and so forth.

As with system passwords, the default credentials for accessing a web-based interface should not be the same across all devices. This is particularly true if the interface is enabled by default as it can be assumed that many end-users will not change the credentials.

- Ensure you can update the software on the device to mitigate vulnerabilities discovered in the future. This consideration especially applies when your device is network-enabled.
- Ensure you remove or disable debugging functionality before producing the final image. For information on how to do this, see the **"Considerations Specific to the OpenEmbedded Build System"** section.
- Ensure you have no network services listening that are not needed.
- Remove any software from the image that is not needed.
- Enable hardware support for secure boot functionality when your device supports this functionality.

7.18.2. Security Flags¶

The Yocto Project has security flags that you can enable that help make your build output more secure. The security flags are in the `meta/conf/distro/include/security_flags.inc` file in your **Source Directory** (e.g. poky).

Note

Depending on the recipe, certain security flags are enabled and disabled by default.

Use the following line in your `local.conf` file or in your custom distribution configuration file to enable the security compiler and linker flags for your build:

```
require conf/distro/include/security_flags.inc
```

7.18.3. Considerations Specific to the OpenEmbedded Build System¶

You can take some steps that are specific to the OpenEmbedded build system to make your images more secure:

- Ensure "debug-tweaks" is not one of your selected `IMAGE_FEATURES`. When creating a new project, the default is to provide you with an initial `local.conf` file that enables this feature using the `EXTRA_IMAGE_FEATURES` variable with the line:

```
EXTRA_IMAGE_FEATURES = "debug-tweaks"
```

To disable that feature, simply comment out that line in your `local.conf` file, or make sure `IMAGE_FEATURES` does not contain "debug-tweaks" before producing your final image. Among other things, leaving this in place sets the root password as blank, which makes logging in for debugging or inspection easy during development but also means anyone can easily log in during production.

- It is possible to set a root password for the image and also to set passwords for any extra users you might add (e.g. administrative or service type users). When you set up passwords for multiple images or users, you should not duplicate passwords.

To set up passwords, use the `extrausers` class, which is the preferred method. For an example on how to set up both root and user passwords, see the "`extrausers.bbclass`" section.

Note

When adding extra user accounts or setting a root password, be cautious about setting the same password on every device. If you do this, and the password you have set is exposed, then every device is now potentially compromised. If you need this access but want to ensure security, consider setting a different, random password for each device. Typically, you do this as a separate step after you deploy the image onto the device.

- Consider enabling a Mandatory Access Control (MAC) framework such as SMACK or SELinux and tuning it appropriately for your device's usage. You can find more information in the `meta-selinux` layer.

7.18.4. Tools for Hardening Your Image¶

The Yocto Project provides tools for making your image more secure. You can find these tools in the `meta-security` layer of the [Yocto Project Source Repositories](#).

7.19. Creating Your Own Distribution¶

When you build an image using the Yocto Project and do not alter any distribution `Metadata`, you are creating a Poky distribution. If you wish to gain more control over package alternative selections, compile-time options, and other low-level configurations, you can create your own distribution.

To create your own distribution, the basic steps consist of creating your own distribution layer, creating your own distribution configuration file, and then adding any needed code and `Metadata` to the layer. The following steps provide some more detail:

- **Create a layer for your new distro:** Create your distribution layer so that you can keep your `Metadata` and code for the distribution separate. It is strongly recommended that you create and use your own layer for configuration and code. Using your own layer as compared to just placing configurations in a `local.conf` configuration file makes it easier to reproduce the same build configuration when using multiple build machines. See the "[Creating a General Layer Using the bitbake-layers Script](#)" section for information on how to quickly set up a layer.
- **Create the distribution configuration file:** The distribution configuration file needs to be created in the `conf/distro` directory of your layer. You need to name it using your distribution name (e.g. `mydistro.conf`).

Note

The `DISTRO` variable in your `local.conf` file determines the name of your distribution.

You can split out parts of your configuration file into include files and then "require" them from within your distribution configuration file. Be sure to place the include files in the `conf/distro/include` directory of your layer. A common example usage of include files would be to separate out the selection of desired version and revisions for individual recipes.

Your configuration file needs to set the following required variables:

```
DISTRO_NAME
DISTRO_VERSION
```

These following variables are optional and you typically set them from the distribution configuration file:

```
DISTRO_FEATURES
DISTRO_EXTRA_RDEPENDS
DISTRO_EXTRA_RRECOMMENDS
TCLIBC
```


Tip

If you want to base your distribution configuration file on the very basic configuration from OE-Core, you can use `conf/distro/defaultsetup.conf` as a reference and just include variables that differ as compared to `defaultsetup.conf`. Alternatively, you can create a distribution configuration file from scratch using the `defaultsetup.conf` file or configuration files from other distributions such as Poky or Angstrom as references.

- **Provide miscellaneous variables:** Be sure to define any other variables for which you want to create a default or enforce as part of the distribution configuration. You can include nearly any variable from the `local.conf` file. The variables you use are not limited to the list in the previous bulleted item.
- **Point to Your distribution configuration file:** In your `local.conf` file in the [Build Directory](#), set your `DISTRO` variable to point to your distribution's configuration file. For example, if your distribution's configuration file is named `mydistro.conf`, then you point to it as follows:

```
DISTRO = "mydistro"
```

- **Add more to the layer if necessary:** Use your layer to hold other information needed for the distribution:
 - Add recipes for installing distro-specific configuration files that are not already installed by another recipe. If you have distro-specific configuration files that are included by an existing recipe, you should add an append file (`.bbappend`) for those. For general information and recommendations on how to add recipes to your layer, see the ["Creating Your Own Layer"](#) and ["Following Best Practices When Creating Layers"](#) sections.
 - Add any image recipes that are specific to your distribution.
 - Add a `psplash` append file for a branded splash screen. For information on append files, see the ["Using .bbappend Files in Your Layer"](#) section.
 - Add any other append files to make custom changes that are specific to individual recipes.

7.20. Creating a Custom Template Configuration Directory¶

If you are producing your own customized version of the build system for use by other users, you might want to customize the message shown by the setup script or you might want to change the template configuration files (i.e. `local.conf` and `bblayers.conf`) that are created in a new build directory.

The OpenEmbedded build system uses the environment variable `TEMPLATECONF` to locate the directory from which it gathers configuration information that ultimately ends up in the [Build Directory](#) `conf` directory. By default, `TEMPLATECONF` is set as follows in the poky repository:

```
TEMPLATECONF=${TEMPLATECONF:-meta-poky/conf}
```

This is the directory used by the build system to find templates from which to build some key configuration files. If you look at this directory, you will see the `bblayers.conf.sample`, `local.conf.sample`, and `conf-notes.txt` files. The build system uses these files to form the respective `bblayers.conf` file, `local.conf` file, and display the list of BitBake targets when running the setup script.

To override these default configuration files with configurations you want used within every new Build Directory, simply set the `TEMPLATECONF` variable to your directory. The `TEMPLATECONF` variable is set in the `.templateconf` file, which is in the top-level [Source Directory](#) folder (e.g. `poky`). Edit the `.templateconf` so that it can locate your directory.

Best practices dictate that you should keep your template configuration directory in your custom distribution layer. For example, suppose you have a layer named `meta-my-layer` located in your home directory and you want your template configuration directory named `myconf`. Changing the `.templateconf` as follows causes the OpenEmbedded build system to look in your directory and base its configuration files on the `*.sample` configuration files it finds. The final configuration files (i.e. `local.conf` and `bblayers.conf`) ultimately still end up in your Build Directory, but they are based on your `*.sample` files.

```
TEMPLATECONF=${TEMPLATECONF:-meta-my-layer/myconf}
```

Aside from the `*.sample` configuration files, the `conf-notes.txt` also resides in the default `meta-poky/conf` directory. The script that sets up the build environment (i.e. [oe-init-build-env](#)) uses this file to display BitBake targets as part of the script output. Customizing this `conf-notes.txt` file is a good way to make sure your list of custom targets appears as part of the script's output.

Here is the default list of targets displayed as a result of running either of the setup scripts:

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

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

Changing the listed common targets is as easy as editing your version of `conf-notes.txt` in your custom template configuration directory and making sure you have `TEMPLATECONF` set to your directory.

7.21. Conserving Disk Space During Builds¶

To help conserve disk space during builds, you can add the following statement to your project's `local.conf` configuration file found in the [Build Directory](#):

```
INHERIT += "rm_work"
```

Adding this statement deletes the work directory used for building a recipe once the recipe is built. For more information on "rm_work", see the [rm_work](#) class in the Yocto Project Reference Manual.

7.22. Working with Packages¶

This section describes a few tasks that involve packages:

- [Excluding packages from an image](#)
- [Incrementing a binary package version](#)
- [Handling optional module packaging](#)
- [Using runtime package management](#)
- [Generating and using signed packages](#)
- [Setting up and running package test \(ptest\)](#)
- [Creating node package manager \(NPM\) packages](#)

7.22.1. Excluding Packages from an Image¶

You might find it necessary to prevent specific packages from being installed into an image. If so, you can use several variables to direct the build system to essentially ignore installing recommended packages or to not install a package at all.

The following list introduces variables you can use to prevent packages from being installed into your image. Each of these variables only works with IPK and RPM package types. Support for Debian packages does not exist. Also, you can use these variables from your `local.conf` file or attach them to a specific image recipe by using a recipe name override. For more detail on the variables, see the descriptions in the Yocto Project Reference Manual's glossary chapter.

- [BAD_RECOMMENDATIONS](#): Use this variable to specify "recommended-only" packages that you do not want installed.
- [NO_RECOMMENDATIONS](#): Use this variable to prevent all "recommended-only" packages from being installed.
- [PACKAGE_EXCLUDE](#): Use this variable to prevent specific packages from being installed regardless of whether they are "recommended-only" or not. You need to realize that the build process could fail with an error when you prevent the installation of a package whose presence is required by an installed package.

7.22.2. Incrementing a Package Version¶

This section provides some background on how binary package versioning is accomplished and presents some of the services, variables, and terminology involved.

In order to understand binary package versioning, you need to consider the following:

- **Binary Package**: The binary package that is eventually built and installed into an image.
- **Binary Package Version**: The binary package version is composed of two components - a version and a revision.

Note

Technically, a third component, the "epoch" (i.e. [PE](#)) is involved but this discussion for the most part ignores [PE](#).

The version and revision are taken from the [PV](#) and [PR](#) variables, respectively.

- [PV](#): The recipe version. [PV](#) represents the version of the software being packaged. Do not confuse [PV](#) with the binary package version.
- [PR](#): The recipe revision.
- [SRCREV](#): The OpenEmbedded build system uses this string to help define the value of [PV](#) when the source code revision needs to be included in it.
- [PR Service](#): A network-based service that helps automate keeping package feeds compatible with existing package manager applications such as RPM, APT, and OPKG.

Whenever the binary package content changes, the binary package version must change. Changing the binary package version is accomplished by changing or "bumping" the [PR](#) and/or [PV](#) values. Increasing these values occurs one of two ways:

- Automatically using a Package Revision Service ([PR Service](#)).
- Manually incrementing the [PR](#) and/or [PV](#) variables.

Given a primary challenge of any build system and its users is how to maintain a package feed that is compatible with existing package manager applications such as RPM, APT, and OPKG, using an automated system is much preferred over a manual system. In either system, the main requirement is that binary package version numbering increases in a linear fashion and that a number of version components exist that support that linear progression. For information on how to ensure package revisioning remains linear, see the ["Automatically Incrementing a Binary Package Revision Number"](#) section.

The following three sections provide related information on the [PR Service](#), the manual method for "bumping" [PR](#) and/or [PV](#), and on how to ensure binary package revisioning remains linear.

7.22.2.1. Working With a PR Service¶

As mentioned, attempting to maintain revision numbers in the `Metadata` is error prone, inaccurate, and causes problems for people submitting recipes. Conversely, the PR Service automatically generates increasing numbers, particularly the revision field, which removes the human element.

Note

For additional information on using a PR Service, you can see the [PR Service](#) wiki page.

The Yocto Project uses variables in order of decreasing priority to facilitate revision numbering (i.e. `PE`, `PV`, and `PR` for epoch, version, and revision, respectively). The values are highly dependent on the policies and procedures of a given distribution and package feed.

Because the OpenEmbedded build system uses "signatures", which are unique to a given build, the build system knows when to rebuild packages. All the inputs into a given task are represented by a signature, which can trigger a rebuild when different. Thus, the build system itself does not rely on the `PR`, `PV`, and `PE` numbers to trigger a rebuild. The signatures, however, can be used to generate these values.

The PR Service works with both `OEBasic` and `OEBasicHash` generators. The value of `PR` bumps when the checksum changes and the different generator mechanisms change signatures under different circumstances.

As implemented, the build system includes values from the PR Service into the `PR` field as an addition using the form ". x" so `r0` becomes `r0.1`, `r0.2` and so forth. This scheme allows existing `PR` values to be used for whatever reasons, which include manual `PR` bumps, should it be necessary.

By default, the PR Service is not enabled or running. Thus, the packages generated are just "self consistent". The build system adds and removes packages and there are no guarantees about upgrade paths but images will be consistent and correct with the latest changes.

The simplest form for a PR Service is for it to exist for a single host development system that builds the package feed (building system). For this scenario, you can enable a local PR Service by setting `PRSERV_HOST` in your `local.conf` file in the [Build Directory](#):

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

Once the service is started, packages will automatically get increasing `PR` values and BitBake takes care of starting and stopping the server.

If you have a more complex setup where multiple host development systems work against a common, shared package feed, you have a single PR Service running and it is connected to each building system. For this scenario, you need to start the PR Service using the `bitbake-prserv` command:

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

In addition to hand-starting the service, you need to update the `local.conf` file of each building system as described earlier so each system points to the server and port.

It is also recommended you use build history, which adds some sanity checks to binary package versions, in conjunction with the server that is running the PR Service. To enable build history, add the following to each building system's `local.conf` file:

```
# It is recommended to activate "buildhistory" for testing the PR service
INHERIT += "buildhistory"
BUILDHISTORY_COMMIT = "1"
```

For information on build history, see the ["Maintaining Build Output Quality"](#) section.

Note

The OpenEmbedded build system does not maintain `PR` information as part of the shared state (`sstate`) packages. If you maintain an `sstate` feed, it's expected that either all your building systems that contribute to the `sstate` feed use a shared PR Service, or you do not run a PR Service on any of your building systems. Having some systems use a PR Service while others do not leads to obvious problems.

For more information on shared state, see the ["Shared State Cache"](#) section in the Yocto Project Overview and Concepts Manual.

7.22.2.2. Manually Bumping PR¶

The alternative to setting up a PR Service is to manually "bump" the `PR` variable.

If a committed change results in changing the package output, then the value of the `PR` variable needs to be increased (or "bumped") as part of that commit. For new recipes you should add the `PR` variable and set its initial value equal to "r0", which is the default. Even though the default value is "r0", the practice of adding it to a new recipe makes it harder to forget to bump the variable when you make changes to the recipe in future.

If you are sharing a common `.inc` file with multiple recipes, you can also use the `INC_PR` variable to ensure that the recipes sharing the `.inc` file are rebuilt when the `.inc` file itself is changed. The `.inc` file must set `INC_PR` (initially to "r0"), and all recipes referring to it should set `PR` to "\${INC_PR}.0" initially, incrementing the last number when the recipe is changed. If the `.inc` file is changed then its `INC_PR` should be incremented.

When upgrading the version of a binary package, assuming the `PV` changes, the `PR` variable should be reset to "r0" (or "\${INC_PR}.0" if you are using `INC_PR`).

Usually, version increases occur only to binary packages. However, if for some reason `PV` changes but does not increase, you can increase the `PE` variable (Package Epoch). The `PE` variable defaults to "0".

Binary package version numbering strives to follow the [Debian Version Field Policy Guidelines](#). These guidelines define how versions are compared and what "increasing" a version means.

7.22.2.3. Automatically Incrementing a Package Version Number¶

When fetching a repository, BitBake uses the `SRCREV` variable to determine the specific source code revision from which to build. You set the `SRCREV` variable to `AUTOREV` to cause the OpenEmbedded build system to automatically use the latest revision of the software:

```
SRCREV = "${AUTOREV}"
```

Furthermore, you need to reference `SRCPV` in `PV` in order to automatically update the version whenever the revision of the source code changes. Here is an example:

```
PV = "1.0+git${SRCPV}"
```

The OpenEmbedded build system substitutes `SRCPV` with the following:

```
AUTOINC+source_code_revision
```

The build system replaces the `AUTOINC` with a number. The number used depends on the state of the PR Service:

- If PR Service is enabled, the build system increments the number, which is similar to the behavior of `PR`. This behavior results in linearly increasing package versions, which is desirable. Here is an example:

```
hello-world-git_0.0+git0+b6558dd387-r0.0_armv7a-neon.ipk
hello-world-git_0.0+git1+dd2f5c3565-r0.0_armv7a-neon.ipk
```

- If PR Service is not enabled, the build system replaces the `AUTOINC` placeholder with zero (i.e. "0"). This results in changing the package version since the source revision is included. However, package versions are not increased linearly. Here is an example:

```
hello-world-git_0.0+git0+b6558dd387-r0.0_armv7a-neon.ipk
hello-world-git_0.0+git0+dd2f5c3565-r0.0_armv7a-neon.ipk
```

In summary, the OpenEmbedded build system does not track the history of binary package versions for this purpose. `AUTOINC`, in this case, is comparable to `PR`. If PR server is not enabled, `AUTOINC` in the package version is simply replaced by "0". If PR server is enabled, the build system keeps track of the package versions and bumps the number when the package revision changes.

7.22.3. Handling Optional Module Packaging¶

Many pieces of software split functionality into optional modules (or plugins) and the plugins that are built might depend on configuration options. To avoid having to duplicate the logic that determines what modules are available in your recipe or to avoid having to package each module by hand, the OpenEmbedded build system provides functionality to handle module packaging dynamically.

To handle optional module packaging, you need to do two things:

- Ensure the module packaging is actually done.
- Ensure that any dependencies on optional modules from other recipes are satisfied by your recipe.

7.22.3.1. Making Sure the Packaging is Done¶

To ensure the module packaging actually gets done, you use the `do_split_packages` function within the `populate_packages` Python function in your recipe. The `do_split_packages` function searches for a pattern of files or directories under a specified path and creates a package for each one it finds by appending to the `PACKAGES` variable and setting the appropriate values for `FILES_packageName`, `RDEPENDS_packageName`, `DESCRIPTION_packageName`, and so forth. Here is an example from the `lighttpd` recipe:

```
python populate_packages_prepend () {
    lighttpd_libdir = d.expand('${libdir}')
    do_split_packages(d, lighttpd_libdir, '^mod_(.*)\.so$',
        'lighttpd-module-%s', 'Lighttpd module for %s',
        extra_depends='')
}
```

The previous example specifies a number of things in the call to `do_split_packages`.

- A directory within the files installed by your recipe through `do_install` in which to search.
- A regular expression used to match module files in that directory. In the example, note the parentheses `()` that mark the part of the expression from which the module name should be derived.
- A pattern to use for the package names.
- A description for each package.
- An empty string for `extra_depends`, which disables the default dependency on the main `lighttpd` package. Thus, if a file in `${libdir}` called `mod_alias.so` is found, a package called `lighttpd-module-alias` is created for it and the `DESCRIPTION` is set to "Lighttpd module for alias".

Often, packaging modules is as simple as the previous example. However, more advanced options exist that you can use within `do_split_packages` to modify its behavior. And, if you need to, you can add more logic by specifying a hook function that is called for each package. It is also perfectly acceptable to call `do_split_packages` multiple times if you have more than one set of modules to package.

For more examples that show how to use `do_split_packages`, see the `connman.inc` file in the `meta/recipes-connectivity/connman/` directory of the poky [source repository](#). You can also find examples in `meta/classes/kernel.bbclass`.

Following is a reference that shows `do_split_packages` mandatory and optional arguments:

```
Mandatory arguments

root
    The path in which to search
file_regex
    Regular expression to match searched files.
    Use parentheses () to mark the part of this
    expression that should be used to derive the
    module name (to be substituted where %s is
    used in other function arguments as noted below)
output_pattern
    Pattern to use for the package names. Must
    include %s.
description
    Description to set for each package. Must
    include %s.

Optional arguments

postinst
    Postinstall script to use for all packages
    (as a string)
recursive
    True to perform a recursive search - default
    False
hook
    A hook function to be called for every match.
    The function will be called with the following
    arguments (in the order listed):

    f
        Full path to the file/directory match
    pkg
        The package name
    file_regex
        As above
    output_pattern
        As above
    modulename
        The module name derived using file_regex

extra_depends
    Extra runtime dependencies (RDEPENDS) to be
    set for all packages. The default value of None
    causes a dependency on the main package
    (${PN}) - if you do not want this, pass empty
    string '' for this parameter.
aux_files_pattern
    Extra item(s) to be added to FILES for each
    package. Can be a single string item or a list
    of strings for multiple items. Must include %s.
postrm
    postrm script to use for all packages (as a
    string)
allow_dirs
    True to allow directories to be matched -
    default False
prepend
    If True, prepend created packages to PACKAGES
    instead of the default False which appends them
match_path
    match file_regex on the whole relative path to
    the root rather than just the file name
aux_files_pattern_verbatim
    Extra item(s) to be added to FILES for each
    package, using the actual derived module name
    rather than converting it to something legal
    for a package name. Can be a single string item
    or a list of strings for multiple items. Must
    include %s.
allow_links
    True to allow symlinks to be matched - default
    False
summary
    Summary to set for each package. Must include %s;
    defaults to description if not set.
```

7.22.3.2. Satisfying Dependencies¶

The second part for handling optional module packaging is to ensure that any dependencies on optional modules from other recipes are satisfied by your recipe. You can be sure these dependencies are satisfied by using the `PACKAGES_DYNAMIC` variable. Here is an example that continues with the `lighttpd` recipe shown earlier:

```
PACKAGES_DYNAMIC = "lighttpd-module-.*"
```

The name specified in the regular expression can of course be anything. In this example, it is `lighttpd-module-` and is specified as the prefix to ensure that any `RDEPENDS` and `RRECOMMENDS` on a package name starting with the prefix are satisfied during build time. If you are using `do_split_packages` as described in the previous section, the value you put in `PACKAGES_DYNAMIC` should correspond to the name pattern specified in the call to `do_split_packages`.

7.22.4. Using Runtime Package Management¶

During a build, BitBake always transforms a recipe into one or more packages. For example, BitBake takes the `bash` recipe and produces a number of packages (e.g. `bash`, `bash-bashbug`, `bash-completion`, `bash-completion-dbg`,

`bash-completion-dev`, `bash-completion-extra`, `bash-debug`, and so forth). Not all generated packages are included in an image.

In several situations, you might need to update, add, remove, or query the packages on a target device at runtime (i.e. without having to generate a new image). Examples of such situations include:

- You want to provide in-the-field updates to deployed devices (e.g. security updates).
- You want to have a fast turn-around development cycle for one or more applications that run on your device.
- You want to temporarily install the "debug" packages of various applications on your device so that debugging can be greatly improved by allowing access to symbols and source debugging.
- You want to deploy a more minimal package selection of your device but allow in-the-field updates to add a larger selection for customization.

In all these situations, you have something similar to a more traditional Linux distribution in that in-field devices are able to receive pre-compiled packages from a server for installation or update. Being able to install these packages on a running, in-field device is what is termed "runtime package management".

In order to use runtime package management, you need a host or server machine that serves up the pre-compiled packages plus the required metadata. You also need package manipulation tools on the target. The build machine is a likely candidate to act as the server. However, that machine does not necessarily have to be the package server. The build machine could push its artifacts to another machine that acts as the server (e.g. Internet-facing). In fact, doing so is advantageous for a production environment as getting the packages away from the development system's build directory prevents accidental overwrites.

A simple build that targets just one device produces more than one package database. In other words, the packages produced by a build are separated out into a couple of different package groupings based on criteria such as the target's CPU architecture, the target board, or the C library used on the target. For example, a build targeting the `qemux86` device produces the following three package databases: `noarch`, `i586`, and `qemux86`. If you wanted your `qemux86` device to be aware of all the packages that were available to it, you would need to point it to each of these databases individually. In a similar way, a traditional Linux distribution usually is configured to be aware of a number of software repositories from which it retrieves packages.

Using runtime package management is completely optional and not required for a successful build or deployment in any way. But if you want to make use of runtime package management, you need to do a couple things above and beyond the basics. The remainder of this section describes what you need to do.

7.22.4.1. Build Considerations¶

This section describes build considerations of which you need to be aware in order to provide support for runtime package management.

When BitBake generates packages, it needs to know what format or formats to use. In your configuration, you use the `PACKAGE_CLASSES` variable to specify the format:

1. Open the `local.conf` file inside your [Build Directory](#) (e.g. `~/poky/build/conf/local.conf`).
2. Select the desired package format as follows:

```
PACKAGE_CLASSES ?= "package_packageformat"
```

where `packageformat` can be "ipk", "rpm", "deb", or "tar" which are the supported package formats.

Note

Because the Yocto Project supports four different package formats, you can set the variable with more than one argument. However, the OpenEmbedded build system only uses the first argument when creating an image or Software Development Kit (SDK).

If you would like your image to start off with a basic package database containing the packages in your current build as well as to have the relevant tools available on the target for runtime package management, you can include "package-management" in the `IMAGE_FEATURES` variable. Including "package-management" in this configuration variable ensures that when the image is assembled for your target, the image includes the currently-known package databases as well as the target-specific tools required for runtime package management to be performed on the target. However, this is not strictly necessary. You could start your image off without any databases but only include the required on-target package tool(s). As an example, you could include "opkg" in your `IMAGE_INSTALL` variable if you are using the IPK package format. You can then initialize your target's package database(s) later once your image is up and running.

Whenever you perform any sort of build step that can potentially generate a package or modify existing package, it is always a good idea to re-generate the package index after the build by using the following command:

```
$ bitbake package-index
```

It might be tempting to build the package and the package index at the same time with a command such as the following:

```
$ bitbake some-package package-index
```

Do not do this as BitBake does not schedule the package index for after the completion of the package you are building. Consequently, you cannot be sure of the package index including information for the package you just built. Thus, be sure to run the package update step separately after building any packages.

You can use the `PACKAGE_FEED_ARCHS`, `PACKAGE_FEED_BASE_PATHS`, and `PACKAGE_FEED_URIS` variables to pre-configure target images to use a package feed. If you do not define these variables, then manual steps as described in the subsequent sections are necessary to configure the target. You should set these variables before building the image in order to produce a correctly configured image.

When your build is complete, your packages reside in the `${TMPDIR}/deploy/packageformat` directory. For example, if `${TMPDIR}` is `tmp` and your selected package type is RPM, then your RPM packages are available in `tmp/deploy/rpm`.

7.22.4.2. Host or Server Machine Setup¶

Although other protocols are possible, a server using HTTP typically serves packages. If you want to use HTTP, then set up and configure a web server such as Apache 2, lighttpd, or SimpleHTTPServer on the machine serving the packages.

To keep things simple, this section describes how to set up a SimpleHTTPServer web server to share package feeds from the developer's machine. Although this server might not be the best for a production environment, the setup is simple and straight forward. Should you want to use a different server more suited for production (e.g. Apache 2, Lighttpd, or Nginx), take the appropriate steps to do so.

From within the build directory where you have built an image based on your packaging choice (i.e. the `PACKAGE_CLASSES` setting), simply start the server. The following example assumes a build directory of `~/poky/build/tmp/depoy/rpm` and a `PACKAGE_CLASSES` setting of "package_rpm":

```
$ cd ~/poky/build/tmp/depoy/rpm
$ python -m SimpleHTTPServer
```

7.22.4.3. Target Setup¶

Setting up the target differs depending on the package management system. This section provides information for RPM, IPK, and DEB.

7.22.4.3.1. Using RPM¶

The [Dandified Packaging Tool](#) (DNF) performs runtime package management of RPM packages. In order to use DNF for runtime package management, you must perform an initial setup on the target machine for cases where the `PACKAGE_FEED_*` variables were not set as part of the image that is running on the target. This means if you built your image and did not use these variables as part of the build and your image is now running on the target, you need to perform the steps in this section if you want to use runtime package management.

Note

For information on the `PACKAGE_FEED_*` variables, see [PACKAGE_FEED_ARCHS](#), [PACKAGE_FEED_BASE_PATHS](#), and [PACKAGE_FEED_URIS](#) in the Yocto Project Reference Manual variables glossary.

On the target, you must inform DNF that package databases are available. You do this by creating a file named `/etc/yum.repos.d/oe-packages.repo` and defining the `oe-packages`.

As an example, assume the target is able to use the following package databases: `all`, `i586`, and `qemux86` from a server named `my.server`. The specifics for setting up the web server are up to you. The critical requirement is that the URIs in the target repository configuration point to the correct remote location for the feeds.

Tip

For development purposes, you can point the web server to the build system's `deploy` directory. However, for production use, it is better to copy the package directories to a location outside of the build area and use that location. Doing so avoids situations where the build system overwrites or changes the `deploy` directory.

When telling DNF where to look for the package databases, you must declare individual locations per architecture or a single location used for all architectures. You cannot do both:

- **Create an Explicit List of Architectures:** Define individual base URLs to identify where each package database is located:

```
[oe-packages]
baseurl=http://my.server/rpm/i586 http://my.server/rpm/qemux86 http://my.server/rpm/all
```

This example informs DNF about individual package databases for all three architectures.

- **Create a Single (Full) Package Index:** Define a single base URL that identifies where a full package database is located:

```
[oe-packages]
baseurl=http://my.server/rpm
```

This example informs DNF about a single package database that contains all the package index information for all supported architectures.

Once you have informed DNF where to find the package databases, you need to fetch them:

```
# dnf makecache
```

DNF is now able to find, install, and upgrade packages from the specified repository or repositories.

Note

See the [DNF documentation](#) for additional information.

7.22.4.3.2. Using IPK¶

The `opkg` application performs runtime package management of IPK packages. You must perform an initial setup for `opkg` on the target machine if the `PACKAGE_FEED_ARCHS`, `PACKAGE_FEED_BASE_PATHS`, and `PACKAGE_FEED_URI` variables have not been set or the target image was built before the variables were set.

The `opkg` application uses configuration files to find available package databases. Thus, you need to create a configuration file inside the `/etc/opkg/` direction, which informs `opkg` of any repository you want to use.

As an example, suppose you are serving packages from a `ipk/` directory containing the `i586`, `all`, and `qemux86` databases through an HTTP server named `my.server`. On the target, create a configuration file (e.g. `my_repo.conf`) inside the `/etc/opkg/` directory containing the following:

```
src/gz all http://my.server/ipk/all
src/gz i586 http://my.server/ipk/i586
src/gz qemux86 http://my.server/ipk/qemux86
```

Next, instruct `opkg` to fetch the repository information:

```
# opkg update
```

The `opkg` application is now able to find, install, and upgrade packages from the specified repository.

7.22.4.3.3. Using DEB

The `apt` application performs runtime package management of DEB packages. This application uses a source list file to find available package databases. You must perform an initial setup for `apt` on the target machine if the `PACKAGE_FEED_ARCHS`, `PACKAGE_FEED_BASE_PATHS`, and `PACKAGE_FEED_URI` variables have not been set or the target image was built before the variables were set.

To inform `apt` of the repository you want to use, you might create a list file (e.g. `my_repo.list`) inside the `/etc/apt/sources.list.d/` directory. As an example, suppose you are serving packages from a `deb/` directory containing the `i586`, `all`, and `qemux86` databases through an HTTP server named `my.server`. The list file should contain:

```
deb http://my.server/deb/all ./
deb http://my.server/deb/i586 ./
deb http://my.server/deb/qemux86 ./
```

Next, instruct the `apt` application to fetch the repository information:

```
# apt-get update
```

After this step, `apt` is able to find, install, and upgrade packages from the specified repository.

7.22.5. Generating and Using Signed Packages

In order to add security to RPM packages used during a build, you can take steps to securely sign them. Once a signature is verified, the OpenEmbedded build system can use the package in the build. If security fails for a signed package, the build system aborts the build.

This section describes how to sign RPM packages during a build and how to use signed package feeds (repositories) when doing a build.

7.22.5.1. Signing RPM Packages

To enable signing RPM packages, you must set up the following configurations in either your `local.config` or `distro.config` file:

```
# Inherit sign_rpm.bbclass to enable signing functionality
INHERIT += " sign_rpm"
# Define the GPG key that will be used for signing.
RPM_GPG_NAME = "key_name"
# Provide passphrase for the key
RPM_GPG_PASSPHRASE = "passphrase"
```

Note

Be sure to supply appropriate values for both `key_name` and `passphrase`

Aside from the `RPM_GPG_NAME` and `RPM_GPG_PASSPHRASE` variables in the previous example, two optional variables related to signing exist:

- **GPG_BIN**: Specifies a `gpg` binary/wrapper that is executed when the package is signed.
- **GPG_PATH**: Specifies the `gpg` home directory used when the package is signed.

7.22.5.2. Processing Package Feeds

In addition to being able to sign RPM packages, you can also enable signed package feeds for IPK and RPM packages.

The steps you need to take to enable signed package feed use are similar to the steps used to sign RPM packages. You must define the following in your `local.config` or `distro.config` file:

```
INHERIT += "sign_package_feed"
PACKAGE_FEED_GPG_NAME = "key_name"
PACKAGE_FEED_GPG_PASSPHRASE_FILE = "path_to_file_containing_passphrase"
```

For signed package feeds, the passphrase must exist in a separate file, which is pointed to by the `PACKAGE_FEED_GPG_PASSPHRASE_FILE` variable. Regarding security, keeping a plain text passphrase out of the configuration is more secure.

Aside from the `PACKAGE_FEED_GPG_NAME` and `PACKAGE_FEED_GPG_PASSPHRASE_FILE` variables, three optional variables related to signed package feeds exist:

- **`GPG_BIN`:** Specifies a `gpg` binary/wrapper that is executed when the package is signed.
- **`GPG_PATH`:** Specifies the `gpg` home directory used when the package is signed.
- **`PACKAGE_FEED_GPG_SIGNATURE_TYPE`:** Specifies the type of `gpg` signature. This variable applies only to RPM and IPK package feeds. Allowable values for the `PACKAGE_FEED_GPG_SIGNATURE_TYPE` are "ASC", which is the default and specifies ascii armored, and "BIN", which specifies binary.

7.22.6. Testing Packages With `pptest`

A Package Test (`pptest`) runs tests against packages built by the OpenEmbedded build system on the target machine. A `pptest` contains at least two items: the actual test, and a shell script (`run-pptest`) that starts the test. The shell script that starts the test must not contain the actual test - the script only starts the test. On the other hand, the test can be anything from a simple shell script that runs a binary and checks the output to an elaborate system of test binaries and data files.

The test generates output in the format used by Automake:

```
result: testname
```

where the result can be `PASS`, `FAIL`, or `SKIP`, and the `testname` can be any identifying string.

For a list of Yocto Project recipes that are already enabled with `pptest`, see the [Pptest](#) wiki page.

Note

A recipe is "pptest-enabled" if it inherits the `pptest` class.

7.22.6.1. Adding `pptest` to Your Build

To add package testing to your build, add the `DISTRO_FEATURES` and `EXTRA_IMAGE_FEATURES` variables to your `local.conf` file, which is found in the [Build Directory](#):

```
DISTRO_FEATURES_append = " pptest"
EXTRA_IMAGE_FEATURES += "pptest-pkgs"
```

Once your build is complete, the `pptest` files are installed into the `/usr/lib/package/pptest` directory within the image, where `package` is the name of the package.

7.22.6.2. Running `pptest`

The `pptest-runner` package installs a shell script that loops through all installed `pptest` test suites and runs them in sequence. Consequently, you might want to add this package to your image.

7.22.6.3. Getting Your Package Ready

In order to enable a recipe to run installed `pptest`s on target hardware, you need to prepare the recipes that build the packages you want to test. Here is what you have to do for each recipe:

- **Be sure the recipe inherits the `pptest` class:** Include the following line in each recipe:

```
inherit pptest
```

- **Create `run-pptest`:** This script starts your test. Locate the script where you will refer to it using `SRC_URI`. Here is an example that starts a test for `dbus`:

```
#!/bin/sh
cd test
make -k runtest-TESTS
```

- **Ensure dependencies are met:** If the test adds build or runtime dependencies that normally do not exist for the package (such as requiring "make" to run the test suite), use the `DEPENDS` and `RDEPENDS` variables in your recipe in order for the package to meet the dependencies. Here is an example where the package has a runtime dependency on "make":

```
RDEPENDS_${PN}-pptest += "make"
```

- **Add a function to build the test suite:** Not many packages support cross-compilation of their test suites. Consequently, you usually need to add a cross-compilation function to the package.

Many packages based on Automake compile and run the test suite by using a single command such as `make check`. However, the host `make check` builds and runs on the same computer, while cross-compiling requires that the package is built on the host but executed for the target architecture (though often, as in the case for `pptest`, the execution occurs on the host). The built version of Automake that ships with the Yocto Project includes a patch that separates building and execution. Consequently, packages that use the unaltered, patched version of `make check` automatically cross-compile.

Regardless, you still must add a `do_compile_pptest` function to build the test suite. Add a function similar to the following to your recipe:

```
do_compile_pptest() {
    oe_runmake buildtest-TESTS
}
```

- **Ensure special configurations are set:** If the package requires special configurations prior to compiling the test code, you must insert a `do_configure_ptest` function into the recipe.
- **Install the test suite:** The `ptest` class automatically copies the file `run-ptest` to the target and then runs `make install-ptest` to run the tests. If this is not enough, you need to create a `do_install_ptest` function and make sure it gets called after the "make install-ptest" completes.

7.22.7. Creating Node Package Manager (NPM) Packages¶

NPM is a package manager for the JavaScript programming language. The Yocto Project supports the NPM [fetcher](#). You can use this fetcher in combination with [devtool](#) to create recipes that produce NPM packages.

Two workflows exist that allow you to create NPM packages using `devtool`: the NPM registry modules method and the NPM project code method.

Note

While it is possible to create NPM recipes manually, using `devtool` is far simpler.

Additionally, some requirements and caveats exist.

7.22.7.1. Requirements and Caveats¶

You need to be aware of the following before using `devtool` to create NPM packages:

- Of the two methods that you can use `devtool` to create NPM packages, the registry approach is slightly simpler. However, you might consider the project approach because you do not have to publish your module in the NPM registry ([npm-registry](#)), which is NPM's public registry.
- Be familiar with [devtool](#).
- The NPM host tools need the native `nodejs-npm` package, which is part of the OpenEmbedded environment. You need to get the package by cloning the <https://github.com/openembedded/meta-openembedded> repository out of GitHub. Be sure to add the path to your local copy to your `bblayers.conf` file.
- `devtool` cannot detect native libraries in module dependencies. Consequently, you must manually add packages to your recipe.
- While deploying NPM packages, `devtool` cannot determine which dependent packages are missing on the target (e.g. the node runtime `nodejs`). Consequently, you need to find out what files are missing and be sure they are on the target.
- Although you might not need NPM to run your node package, it is useful to have NPM on your target. The NPM package name is `nodejs-npm`.

7.22.7.2. Using the Registry Modules Method¶

This section presents an example that uses the `cute-files` module, which is a file browser web application.

Note

You must know the `cute-files` module version.

The first thing you need to do is use `devtool` and the NPM fetcher to create the recipe:

```
$ devtool add "npm://registry.npmjs.org;name=cute-files;version=1.0.2"
```

The `devtool add` command runs `recipetool create` and uses the same fetch URI to download each dependency and capture license details where possible. The result is a generated recipe.

The recipe file is fairly simple and contains every license that `recipetool` finds and includes the licenses in the recipe's `LIC_FILES_CHKSUM` variables. You need to examine the variables and look for those with "unknown" in the `LICENSE` field. You need to track down the license information for "unknown" modules and manually add the information to the recipe.

`recipetool` creates "shrinkwrap" and "lockdown" files for your recipe. Shrinkwrap files capture the version of all dependent modules. Many packages do not provide shrinkwrap files. `recipetool` create a shrinkwrap file as it runs. You can replace the shrinkwrap file with your own file by setting the `NPM_SHRINKWRAP` variable.

Lockdown files contain the checksum for each module to determine if your users download the same files when building with a recipe. Lockdown files ensure that dependencies have not been changed and that your NPM registry is still providing the same file.

Note

A package is created for each sub-module. This policy is the only practical way to have the licenses for all of the dependencies represented in the license manifest of the image.

The `devtool edit-recipe` command lets you take a look at the recipe:

```
$ devtool edit-recipe cute-files
SUMMARY = "Turn any folder on your computer into a cute file browser, available on the local network."
LICENSE = "BSD-3-Clause & Unknown & MIT & ISC"
LIC_FILES_CHKSUM = "file://LICENSE;md5=71d98c0a1db42956787b1909c74a86ca \
file://node_modules/content-disposition/LICENSE;md5=c6e0ce1e688c5ff16db06b7259e9cd20 \
file://node_modules/express/LICENSE;md5=5513c00a5c36cd361da863dd9aa8875d \
```

```

...
SRC_URI = "npm://registry.npmjs.org;name=cute-files;version=${PV}"
NPM_SHRINKWRAP := "${THISDIR}/${PN}/npm-shrinkwrap.json"
NPM_LOCKDOWN := "${THISDIR}/${PN}/lockdown.json"
inherit npm
# Must be set after inherit npm since that itself sets S
S = "${WORKDIR}/npmkg"

LICENSE_${PN}-content-disposition = "MIT"
...
LICENSE_${PN}-express = "MIT"
LICENSE_${PN} = "MIT"

```

Three key points exist in the previous example:

- `SRC_URI` uses the NPM scheme so that the NPM fetcher is used.
- `recipetool` collects all the license information. If a sub-module's license is unavailable, the sub-module's name appears in the comments.
- The `inherit npm` statement causes the `npm` class to package up all the modules.

You can run the following command to build the `cute-files` package:

```
$ devtool build cute-files
```

Remember that `nodejs` must be installed on the target before your package.

Assuming 192.168.7.2 for the target's IP address, use the following command to deploy your package:

```
$ devtool deploy-target -s cute-files root@192.168.7.2
```

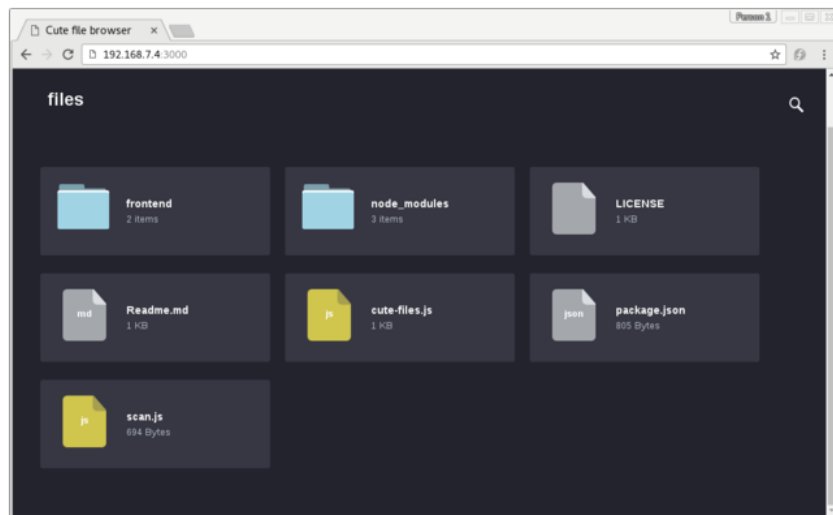
Once the package is installed on the target, you can test the application:

Note

Because of a know issue, you cannot simply run `cute-files` as you would if you had run `npm install`.

```
$ cd /usr/lib/node_modules/cute-files
$ node cute-files.js
```

On a browser, go to `http://192.168.7.2:3000` and you see the following:



You can find the recipe in `workspace/recipes/cute-files`. You can use the recipe in any layer you choose.

7.22.7.3. Using the NPM Projects Code Method¶

Although it is useful to package modules already in the NPM registry, adding `node.js` projects under development is a more common developer use case.

This section covers the NPM projects code method, which is very similar to the "registry" approach described in the previous section. In the NPM projects method, you provide `devtool` with an URL that points to the source files.

Replicating the same example, (i.e. `cute-files`) use the following command:

```
$ devtool add https://github.com/martinaglv/cute-files.git
```

The recipe this command generates is very similar to the recipe created in the previous section. However, the `SRC_URI` looks like the following:

```

SRC_URI = "git://github.com/martinaglv/cute-files.git;protocol=https \
          npm://registry.npmjs.org;name=commander;version=2.9.0;subdir=node_modules/commander \

```

```

npm://registry.npmjs.org;name=express;version=4.14.0;subdir=node_modules/express \
npm://registry.npmjs.org;name=content-disposition;version=0.3.0;subdir=node_modules/content-disposition \
"

```

In this example, the main module is taken from the Git repository and dependents are taken from the NPM registry. Other than those differences, the recipe is basically the same between the two methods. You can build and deploy the package exactly as described in the previous section that uses the registry modules method.

7.23. Efficiently Fetching Source Files During a Build¶

The OpenEmbedded build system works with source files located through the `SRC_URI` variable. When you build something using BitBake, a big part of the operation is locating and downloading all the source tarballs. For images, downloading all the source for various packages can take a significant amount of time.

This section shows you how you can use mirrors to speed up fetching source files and how you can pre-fetch files all of which leads to more efficient use of resources and time.

7.23.1. Setting up Effective Mirrors¶

A good deal that goes into a Yocto Project build is simply downloading all of the source tarballs. Maybe you have been working with another build system (OpenEmbedded or Angstrom) for which you have built up a sizable directory of source tarballs. Or, perhaps someone else has such a directory for which you have read access. If so, you can save time by adding statements to your configuration file so that the build process checks local directories first for existing tarballs before checking the Internet.

Here is an efficient way to set it up in your `local.conf` file:

```

SOURCE_MIRROR_URL ?= "file:///home/you/your-download-dir/"
INHERIT += "own-mirrors"
BB_GENERATE_MIRROR_TARBALLS = "1"
# BB_NO_NETWORK = "1"

```

In the previous example, the `BB_GENERATE_MIRROR_TARBALLS` variable causes the OpenEmbedded build system to generate tarballs of the Git repositories and store them in the `DL_DIR` directory. Due to performance reasons, generating and storing these tarballs is not the build system's default behavior.

You can also use the `PREMIRRORS` variable. For an example, see the variable's glossary entry in the Yocto Project Reference Manual.

7.23.2. Getting Source Files and Suppressing the Build¶

Another technique you can use to ready yourself for a successive string of build operations, is to pre-fetch all the source files without actually starting a build. This technique lets you work through any download issues and ultimately gathers all the source files into your download directory `build/downloads`, which is located with `DL_DIR`.

Use the following BitBake command form to fetch all the necessary sources without starting the build:

```
$ bitbake -c target runall="fetch"
```

This variation of the BitBake command guarantees that you have all the sources for that BitBake target should you disconnect from the Internet and want to do the build later offline.

7.24. Selecting an Initialization Manager¶

By default, the Yocto Project uses SysVinit as the initialization manager. However, support also exists for systemd, which is a full replacement for init with parallel starting of services, reduced shell overhead and other features that are used by many distributions.

Within the system, SysVinit treats system components as services. These services are maintained as shell scripts stored in the `/etc/init.d/` directory. Services organize into different run levels. This organization is maintained by putting links to the services in the `/etc/rcN.d/` directories, where *N* is one of the following options: "S", "0", "1", "2", "3", "4", "5", or "6".

Note

Each runlevel has a dependency on the previous runlevel. This dependency allows the services to work properly.

In comparison, systemd treats components as units. Using units is a broader concept as compared to using a service. A unit includes several different types of entities. Service is one of the types of entities. The runlevel concept in SysVinit corresponds to the concept of a target in systemd, where target is also a type of supported unit.

In a SysVinit-based system, services load sequentially (i.e. one by one) during and parallelization is not supported. With systemd, services start in parallel. Needless to say, the method can have an impact on system startup performance.

If you want to use SysVinit, you do not have to do anything. But, if you want to use systemd, you must take some steps as described in the following sections.

7.24.1. Using systemd Exclusively¶

Set these variables in your distribution configuration file as follows:

```

DISTRO_FEATURES_append = " systemd"
VIRTUAL-RUNTIME_init_manager = "systemd"

```

You can also prevent the SysVinit distribution feature from being automatically enabled as follows:

```
DISTRO_FEATURES_BACKFILL_CONSIDERED = "sysvinit"
```

Doing so removes any redundant SysVinit scripts.

To remove initscripts from your image altogether, set this variable also:

```
VIRTUAL-RUNTIME_initscripts = ""
```

For information on the backfill variable, see [DISTRO_FEATURES_BACKFILL_CONSIDERED](#).

7.24.2. Using systemd for the Main Image and Using SysVinit for the Rescue Image¶

Set these variables in your distribution configuration file as follows:

```
DISTRO_FEATURES_append = " systemd"
VIRTUAL-RUNTIME_init_manager = "systemd"
```

Doing so causes your main image to use the `packagegroup-core-boot.bb` recipe and `systemd`. The rescue/minimal image cannot use this package group. However, it can install SysVinit and the appropriate packages will have support for both `systemd` and SysVinit.

7.25. Selecting a Device Manager¶

The Yocto Project provides multiple ways to manage the device manager (`/dev`):

- **Persistent and Pre-Populated `/dev`:** For this case, the `/dev` directory is persistent and the required device nodes are created during the build.
- **Use `devtmpfs` with a Device Manager:** For this case, the `/dev` directory is provided by the kernel as an in-memory file system and is automatically populated by the kernel at runtime. Additional configuration of device nodes is done in user space by a device manager like `udev` or `busybox-mdev`.

7.25.1. Using Persistent and Pre-Populated `/dev`¶

To use the static method for device population, you need to set the `USE_DEVFS` variable to "0" as follows:

```
USE_DEVFS = "0"
```

The content of the resulting `/dev` directory is defined in a Device Table file. The `IMAGE_DEVICE_TABLES` variable defines the Device Table to use and should be set in the machine or distro configuration file. Alternatively, you can set this variable in your `local.conf` configuration file.

If you do not define the `IMAGE_DEVICE_TABLES` variable, the default `device_table-minimal.txt` is used:

```
IMAGE_DEVICE_TABLES = "device_table-mymachine.txt"
```

The population is handled by the `makedevs` utility during image creation:

7.25.2. Using `devtmpfs` and a Device Manager¶

To use the dynamic method for device population, you need to use (or be sure to set) the `USE_DEVFS` variable to "1", which is the default:

```
USE_DEVFS = "1"
```

With this setting, the resulting `/dev` directory is populated by the kernel using `devtmpfs`. Make sure the corresponding kernel configuration variable `CONFIG_DEVTMPFS` is set when building you build a Linux kernel.

All devices created by `devtmpfs` will be owned by `root` and have permissions `0600`.

To have more control over the device nodes, you can use a device manager like `udev` or `busybox-mdev`. You choose the device manager by defining the `VIRTUAL-RUNTIME_dev_manager` variable in your machine or distro configuration file. Alternatively, you can set this variable in your `local.conf` configuration file:

```
VIRTUAL-RUNTIME_dev_manager = "udev"

# Some alternative values
# VIRTUAL-RUNTIME_dev_manager = "busybox-mdev"
# VIRTUAL-RUNTIME_dev_manager = "systemd"
```

7.26. Using an External SCM¶

If you're working on a recipe that pulls from an external Source Code Manager (SCM), it is possible to have the OpenEmbedded build system notice new recipe changes added to the SCM and then build the resulting packages that depend on the new recipes by using the latest versions. This only works for SCMs from which it is possible to get a sensible revision number for changes. Currently, you can do this with Apache Subversion (SVN), Git, and Bazaar (BZR) repositories.

To enable this behavior, the `PV` of the recipe needs to reference `SRCPV`. Here is an example:

```
PV = "1.2.3+git${SRCPV}"
```

Then, you can add the following to your `local.conf`:

```
SRCREV_pn-PN = "${AUTOREV}"
```

`PN` is the name of the recipe for which you want to enable automatic source revision updating.

If you do not want to update your local configuration file, you can add the following directly to the recipe to finish enabling the feature:

```
SRCREV = "${AUTOREV}"
```

The Yocto Project provides a distribution named `poky-bleeding`, whose configuration file contains the line:

```
require conf/distro/include/poky-floating-revisions.inc
```

This line pulls in the listed include file that contains numerous lines of exactly that form:

```
#SRCREV_pn-opkg-native ?= "${AUTOREV}"
#SRCREV_pn-opkg-sdk ?= "${AUTOREV}"
#SRCREV_pn-opkg ?= "${AUTOREV}"
#SRCREV_pn-opkg-utils-native ?= "${AUTOREV}"
#SRCREV_pn-opkg-utils ?= "${AUTOREV}"
SRCREV_pn-gconf-dbus ?= "${AUTOREV}"
SRCREV_pn-matchbox-common ?= "${AUTOREV}"
SRCREV_pn-matchbox-config-gtk ?= "${AUTOREV}"
SRCREV_pn-matchbox-desktop ?= "${AUTOREV}"
SRCREV_pn-matchbox-keyboard ?= "${AUTOREV}"
SRCREV_pn-matchbox-panel-2 ?= "${AUTOREV}"
SRCREV_pn-matchbox-themes-extra ?= "${AUTOREV}"
SRCREV_pn-matchbox-terminal ?= "${AUTOREV}"
SRCREV_pn-matchbox-wm ?= "${AUTOREV}"
SRCREV_pn-settings-daemon ?= "${AUTOREV}"
SRCREV_pn-screenshot ?= "${AUTOREV}"
.
.
.
```

These lines allow you to experiment with building a distribution that tracks the latest development source for numerous packages.

Caution

The `poky-bleeding` distribution is not tested on a regular basis. Keep this in mind if you use it.

7.27. Creating a Read-Only Root Filesystem

Suppose, for security reasons, you need to disable your target device's root filesystem's write permissions (i.e. you need a read-only root filesystem). Or, perhaps you are running the device's operating system from a read-only storage device. For either case, you can customize your image for that behavior.

Note

Supporting a read-only root filesystem requires that the system and applications do not try to write to the root filesystem. You must configure all parts of the target system to write elsewhere, or to gracefully fail in the event of attempting to write to the root filesystem.

7.27.1. Creating the Root Filesystem

To create the read-only root filesystem, simply add the "read-only-rootfs" feature to your image. Using either of the following statements in your image recipe or from within the `local.conf` file found in the [Build Directory](#) causes the build system to create a read-only root filesystem:

```
IMAGE_FEATURES = "read-only-rootfs"
```

or

```
EXTRA_IMAGE_FEATURES += "read-only-rootfs"
```

For more information on how to use these variables, see the ["Customizing Images Using Custom IMAGE_FEATURES and EXTRA_IMAGE_FEATURES"](#) section. For information on the variables, see [IMAGE_FEATURES](#) and [EXTRA_IMAGE_FEATURES](#).

7.27.2. Post-Installation Scripts

It is very important that you make sure all post-Installation (`pkg_postinst`) scripts for packages that are installed into the image can be run at the time when the root filesystem is created during the build on the host system. These scripts cannot attempt to run during first-boot on the target device. With the "read-only-rootfs" feature enabled, the build system checks during root filesystem creation to make sure all post-installation scripts succeed. If any of these scripts still need to be run after the root filesystem is created, the build immediately fails. These build-time checks ensure that the build fails rather than the target device fails later during its initial boot operation.

Most of the common post-installation scripts generated by the build system for the out-of-the-box Yocto Project are engineered so that they can run during root filesystem creation (e.g. post-installation scripts for caching fonts). However, if you create and add custom scripts, you need to be sure they can be run during this file system creation.

Here are some common problems that prevent post-installation scripts from running during root filesystem creation:

- **Not using `$D` in front of absolute paths:** The build system defines `$D` when the root filesystem is created. Furthermore, `$D` is blank when the script is run on the target device. This implies two purposes for `$D`: ensuring paths are valid in both the host and target environments, and checking to determine which environment is being used as a method for taking appropriate actions.
- **Attempting to run processes that are specific to or dependent on the target architecture:** You can work around these attempts by using native tools, which run on the host system, to accomplish the same tasks, or by alternatively running the processes under QEMU, which has the `qemu_run_binary` function. For more information, see the `qemu` class.

7.27.3. Areas With Write Access¶

With the "read-only-rootfs" feature enabled, any attempt by the target to write to the root filesystem at runtime fails. Consequently, you must make sure that you configure processes and applications that attempt these types of writes do so to directories with write access (e.g. `/tmp` or `/var/run`).

7.28. Maintaining Build Output Quality¶

Many factors can influence the quality of a build. For example, if you upgrade a recipe to use a new version of an upstream software package or you experiment with some new configuration options, subtle changes can occur that you might not detect until later. Consider the case where your recipe is using a newer version of an upstream package. In this case, a new version of a piece of software might introduce an optional dependency on another library, which is auto-detected. If that library has already been built when the software is building, the software will link to the built library and that library will be pulled into your image along with the new software even if you did not want the library.

The `buildhistory` class exists to help you maintain the quality of your build output. You can use the class to highlight unexpected and possibly unwanted changes in the build output. When you enable build history, it records information about the contents of each package and image and then commits that information to a local Git repository where you can examine the information.

The remainder of this section describes the following:

- How you can enable and disable build history
- How to understand what the build history contains
- How to limit the information used for build history
- How to examine the build history from both a command-line and web interface

7.28.1. Enabling and Disabling Build History¶

Build history is disabled by default. To enable it, add the following `INHERIT` statement and set the `BUILDHISTORY_COMMIT` variable to "1" at the end of your `conf/local.conf` file found in the [Build Directory](#):

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

Enabling build history as previously described causes the OpenEmbedded build system to collect build output information and commit it as a single commit to a local [Git](#) repository.

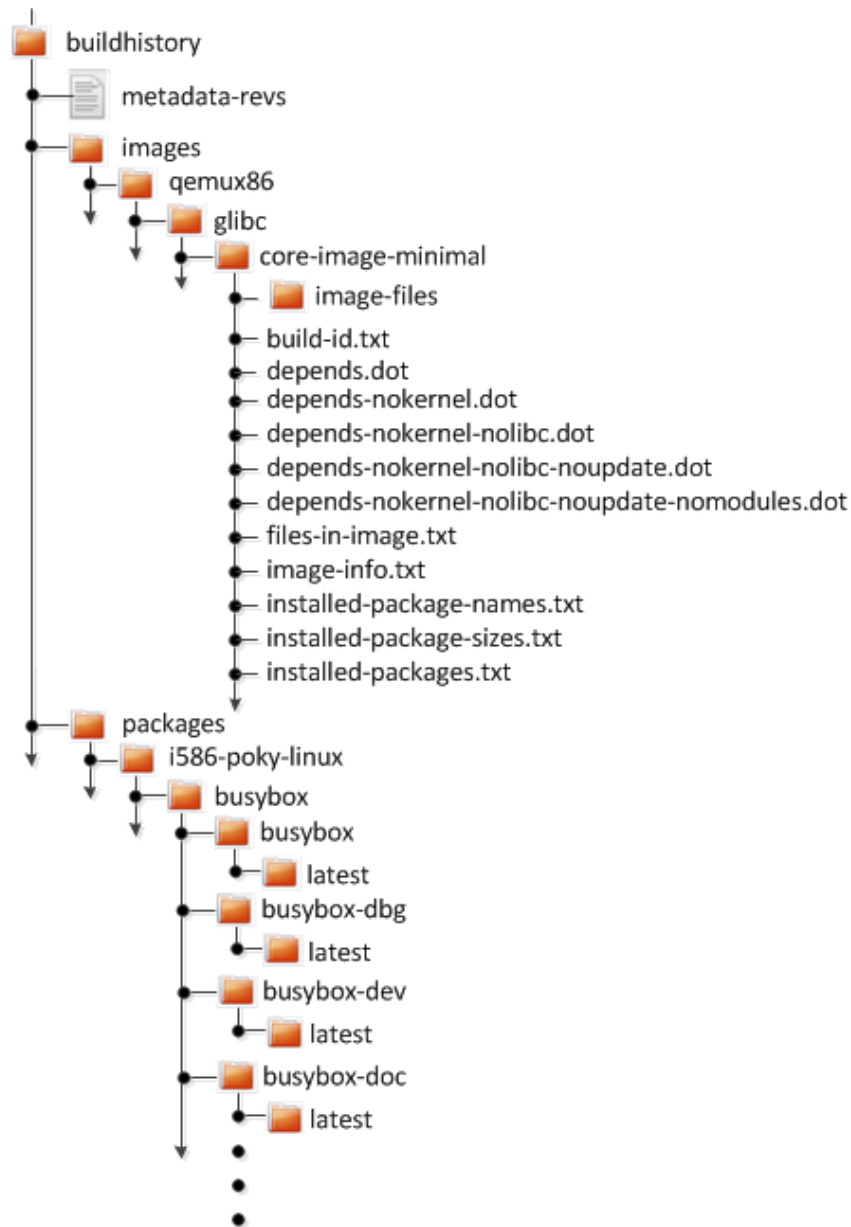
Note

Enabling build history increases your build times slightly, particularly for images, and increases the amount of disk space used during the build.

You can disable build history by removing the previous statements from your `conf/local.conf` file.

7.28.2. Understanding What the Build History Contains¶

Build history information is kept in `${TOPDIR}/buildhistory` in the Build Directory as defined by the `BUILDHISTORY_DIR` variable. The following is an example abbreviated listing:



At the top level, a `metadata-revs` file exists that lists the revisions of the repositories for the enabled layers when the build was produced. The rest of the data splits into separate `packages`, `images` and `sdk` directories, the contents of which are described as follows.

7.28.2.1. Build History Package Information¶

The history for each package contains a text file that has name-value pairs with information about the package. For example, `buildhistory/packages/i586-poky-linux/busybox/busybox/latest` contains the following:

```

PV = 1.22.1
PR = r32
RPROVIDES =
RDEPENDS = glibc (>= 2.20) update-alternatives-opkg
RRECOMMENDS = busybox-syslog busybox-udhcpd update-rc.d
PKGSIZE = 540168
FILES = /usr/bin/* /usr/sbin/* /usr/lib/busybox/* /usr/lib/lib*.so.* \
        /etc/com/var/bin/* /sbin/* /lib/*.* /lib/udev/rules.d \
        /usr/lib/udev/rules.d /usr/share/busybox /usr/lib/busybox/* \
        /usr/share/pixmaps /usr/share/applications /usr/share/idl \
        /usr/share/omf /usr/share/sounds /usr/lib/bonobo/servers
FILELIST = /bin/busybox /bin/busybox.nosuid /bin/busybox.suid /bin/sh \
        /etc/busybox.links.nosuid /etc/busybox.links.suid

```

Most of these name-value pairs correspond to variables used to produce the package. The exceptions are `FILELIST`, which is the actual list of files in the package, and `PKGSIZE`, which is the total size of files in the package in bytes.

A file also exists that corresponds to the recipe from which the package came (e.g. `buildhistory/packages/i586-poky-linux/busybox/latest`):

```

PV = 1.22.1
PR = r32
DEPENDS = initscripts kern-tools-native update-rc.d-native \
        virtual/i586-poky-linux-compilerlibs virtual/i586-poky-linux-gcc \
        virtual/libc virtual/update-alternatives
PACKAGES = busybox-ptest busybox-httpd busybox-udhcpd busybox-udhcpd \
        busybox-syslog busybox-mdev busybox-hwclock busybox-dbg \
        busybox-staticdev busybox-dev busybox-doc busybox-locale busybox

```

Finally, for those recipes fetched from a version control system (e.g., Git), a file exists that lists source revisions that are specified in the recipe and lists the actual revisions used during the build. Listed and actual revisions might differ when `SRCREV` is set to `${AUTOREV}`. Here is an example assuming `buildhistory/packages/qemux86-poky-linux/linux-yocto/latest_srcrev`:

```
# SRCREV_machine = "38cd560d5022ed2dbd1ab0dca9642e47c98a0aa1"
SRCREV_machine = "38cd560d5022ed2dbd1ab0dca9642e47c98a0aa1"
# SRCREV_meta = "a227f20eff056e511d504b2e490f3774ab260d6f"
SRCREV_meta = "a227f20eff056e511d504b2e490f3774ab260d6f"
```

You can use the `buildhistory-collect-srcrevs` command with the `-a` option to collect the stored `SRCREV` values from build history and report them in a format suitable for use in global configuration (e.g., `local.conf` or a distro include file) to override floating `AUTOREV` values to a fixed set of revisions. Here is some example output from this command:

```
$ buildhistory-collect-srcrevs -a
# i586-poky-linux
SRCREV_pn-glibc = "b8079dd0d360648e4e8de48656c5c38972621072"
SRCREV_pn-glibc-initial = "b8079dd0d360648e4e8de48656c5c38972621072"
SRCREV_pn-opkg-utils = "53274f087565fd45d8452c5367997ba6a682a37a"
SRCREV_pn-kmod = "fd56638aed3fe147015bfa10ed4a5f7491303cb4"
# x86_64-linux
SRCREV_pn-gtk-doc-stub-native = "1dea266593edb766d6d898c79451ef193eb17cfa"
SRCREV_pn-dtc-native = "65cc4d2748a2c2e6f27f1cf39e07a5dbabd80ebf"
SRCREV_pn-update-rc.d-native = "eca680ddf28d024954895f59a241a622dd575c11"
SRCREV_pn-glibc_pn-cross-localedef-native = "b8079dd0d360648e4e8de48656c5c38972621072"
SRCREV_pn-localedef_pn-cross-localedef-native = "c83367348d39dad7ba018990bdfdaaef8e9ed3"
SRCREV_pn-prelink-native = "faa069deec99bf61418d0bab831c83d7c1b797ca"
SRCREV_pn-opkg-utils-native = "53274f087565fd45d8452c5367997ba6a682a37a"
SRCREV_pn-kern-tools-native = "23345b8846fe4bd167efdf1bd8a1224b2ba9a5ff"
SRCREV_pn-kmod-native = "fd56638aed3fe147015bfa10ed4a5f7491303cb4"
# qemux86-poky-linux
SRCREV_machine_pn-linux-yocto = "38cd560d5022ed2dbd1ab0dca9642e47c98a0aa1"
SRCREV_meta_pn-linux-yocto = "a227f20eff056e511d504b2e490f3774ab260d6f"
# all-poky-linux
SRCREV_pn-update-rc.d = "eca680ddf28d024954895f59a241a622dd575c11"
```

Note

Here are some notes on using the `buildhistory-collect-srcrevs` command:

- By default, only values where the `SRCREV` was not hardcoded (usually when `AUTOREV` is used) are reported. Use the `-a` option to see all `SRCREV` values.
- The output statements might not have any effect if overrides are applied elsewhere in the build system configuration. Use the `-f` option to add the `forcevariable` override to each output line if you need to work around this restriction.
- The script does apply special handling when building for multiple machines. However, the script does place a comment before each set of values that specifies which triplet to which they belong as previously shown (e.g., `i586-poky-linux`).

7.28.2.2. Build History Image Information

The files produced for each image are as follows:

- `image-files`: A directory containing selected files from the root filesystem. The files are defined by `BUILDHISTORY_IMAGE_FILES`.
- `build-id.txt`: Human-readable information about the build configuration and metadata source revisions. This file contains the full build header as printed by BitBake.
- `*.dot`: Dependency graphs for the image that are compatible with `graphviz`.
- `files-in-image.txt`: A list of files in the image with permissions, owner, group, size, and symlink information.
- `image-info.txt`: A text file containing name-value pairs with information about the image. See the following listing example for more information.
- `installed-package-names.txt`: A list of installed packages by name only.
- `installed-package-sizes.txt`: A list of installed packages ordered by size.
- `installed-packages.txt`: A list of installed packages with full package filenames.

Note

Installed package information is able to be gathered and produced even if package management is disabled for the final image.

Here is an example of `image-info.txt`:

```
DISTRO = poky
DISTRO_VERSION = 1.7
USER_CLASSES = buildstats image-mklibs image-prelink
IMAGE_CLASSES = image_types
IMAGE_FEATURES = debug-tweaks
IMAGE_LINGUAS =
IMAGE_INSTALL = packagegroup-core-boot run-postinsts
BAD_RECOMMENDATIONS =
```

```

NO_RECOMMENDATIONS =
PACKAGE_EXCLUDE =
ROOTFS_POSTPROCESS_COMMAND = write_package_manifest; license_create_manifest; \
    write_image_manifest ; buildhistory_list_installed_image ; \
    buildhistory_get_image_installed ; ssh_allow_empty_password ; \
    postinst_enable_logging; rootfs_update_timestamp ; ssh_disable_dns_lookup ;
IMAGE_POSTPROCESS_COMMAND = buildhistory_get_imageinfo ;
IMAGESIZE = 6900

```

Other than `IMAGESIZE`, which is the total size of the files in the image in Kbytes, the name-value pairs are variables that may have influenced the content of the image. This information is often useful when you are trying to determine why a change in the package or file listings has occurred.

7.28.2.3. Using Build History to Gather Image Information Only¶

As you can see, build history produces image information, including dependency graphs, so you can see why something was pulled into the image. If you are just interested in this information and not interested in collecting specific package or SDK information, you can enable writing only image information without any history by adding the following to your `conf/local.conf` file found in the [Build Directory](#):

```

INHERIT += "buildhistory"
BUILDHISTORY_COMMIT = "0"
BUILDHISTORY_FEATURES = "image"

```

Here, you set the `BUILDHISTORY_FEATURES` variable to use the image feature only.

7.28.2.4. Build History SDK Information¶

Build history collects similar information on the contents of SDKs (e.g. `bitbake -c populate_sdk imagedname`) as compared to information it collects for images. Furthermore, this information differs depending on whether an extensible or standard SDK is being produced.

The following list shows the files produced for SDKs:

- `files-in-sdk.txt`: A list of files in the SDK with permissions, owner, group, size, and symlink information. This list includes both the host and target parts of the SDK.
- `sdk-info.txt`: A text file containing name-value pairs with information about the SDK. See the following listing example for more information.
- `sstate-task-sizes.txt`: A text file containing name-value pairs with information about task group sizes (e.g. `do_populate_sysroot` tasks have a total size). The `sstate-task-sizes.txt` file exists only when an extensible SDK is created.
- `sstate-package-sizes.txt`: A text file containing name-value pairs with information for the shared-state packages and sizes in the SDK. The `sstate-package-sizes.txt` file exists only when an extensible SDK is created.
- `sdk-files`: A folder that contains copies of the files mentioned in `BUILDHISTORY_SDK_FILES` if the files are present in the output. Additionally, the default value of `BUILDHISTORY_SDK_FILES` is specific to the extensible SDK although you can set it differently if you would like to pull in specific files from the standard SDK.

The default files are `conf/local.conf`, `conf/bblayers.conf`, `conf/auto.conf`, `conf/locked-sigs.inc`, and `conf/devtool.conf`. Thus, for an extensible SDK, these files get copied into the `sdk-files` directory.

- The following information appears under each of the `host` and `target` directories for the portions of the SDK that run on the host and on the target, respectively:

Note

The following files for the most part are empty when producing an extensible SDK because this type of SDK is not constructed from packages as is the standard SDK.

- `depends.dot`: Dependency graph for the SDK that is compatible with `graphviz`.
- `installed-package-names.txt`: A list of installed packages by name only.
- `installed-package-sizes.txt`: A list of installed packages ordered by size.
- `installed-packages.txt`: A list of installed packages with full package filenames.

Here is an example of `sdk-info.txt`:

```

DISTRO = poky
DISTRO_VERSION = 1.3+snapshot-20130327
SDK_NAME = poky-glibc-i686-arm
SDK_VERSION = 1.3+snapshot
SDKMACHINE =
SDKIMAGE_FEATURES = dev-pkgs dbg-pkgs
BAD_RECOMMENDATIONS =
SDKSIZE = 352712

```

Other than `SDKSIZE`, which is the total size of the files in the SDK in Kbytes, the name-value pairs are variables that might have influenced the content of the SDK. This information is often useful when you are trying to determine why a change in the package or file listings has occurred.

7.28.2.5. Examining Build History Information¶

You can examine build history output from the command line or from a web interface.

To see any changes that have occurred (assuming you have `BUILDHISTORY_COMMIT = "1"`), you can simply use any Git command that allows you to view the history of a repository. Here is one method:

```
$ git log -p
```

You need to realize, however, that this method does show changes that are not significant (e.g. a package's size changing by a few bytes).

A command-line tool called `buildhistory-diff` does exist, though, that queries the Git repository and prints just the differences that might be significant in human-readable form. Here is an example:

```
$ ~/poky/poky/scripts/buildhistory-diff . HEAD^
Changes to images/qemux86_64/glibc/core-image-minimal (files-in-image.txt):
  /etc/anotherpkg.conf was added
  /sbin/anotherpkg was added
  * (installed-package-names.txt):
  * anotherpkg was added
Changes to images/qemux86_64/glibc/core-image-minimal (installed-package-names.txt):
  anotherpkg was added
packages/qemux86_64-poky-linux/v86d: PACKAGES: added "v86d-extras"
  * PR changed from "r0" to "r1"
  * PV changed from "0.1.10" to "0.1.12"
packages/qemux86_64-poky-linux/v86d/v86d: PKGSIZE changed from 110579 to 144381 (+30%)
  * PR changed from "r0" to "r1"
  * PV changed from "0.1.10" to "0.1.12"
```

Note

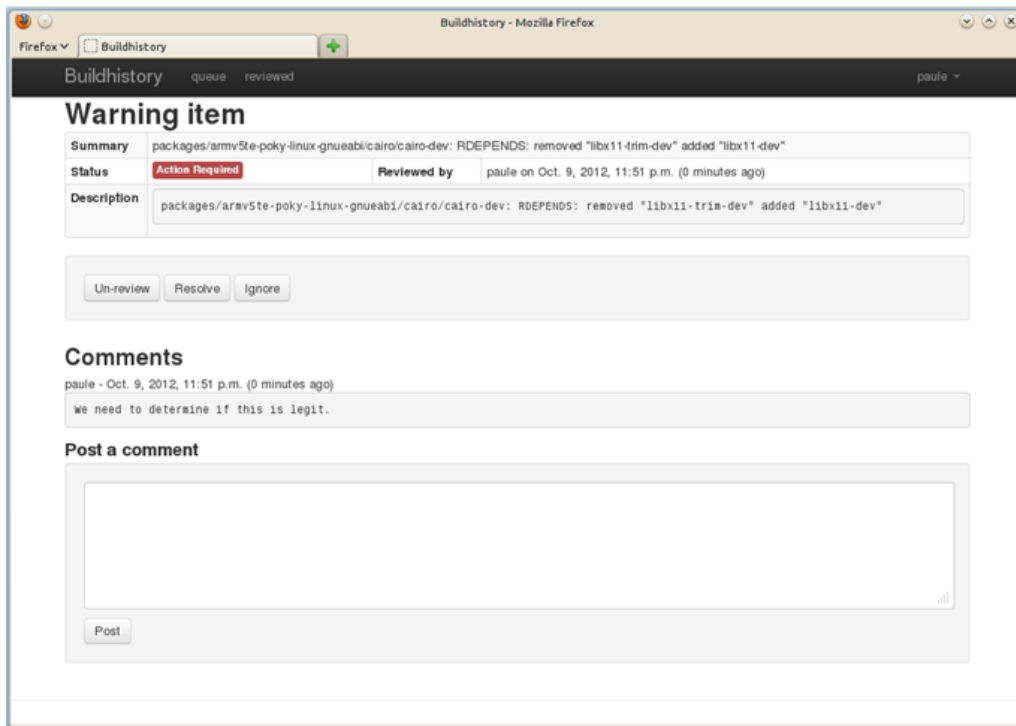
The `buildhistory-diff` tool requires the `GitPython` package. Be sure to install it using `Pip3` as follows:

```
$ pip3 install GitPython --user
```

Alternatively, you can install `python3-git` using the appropriate distribution package manager (e.g. `apt-get`, `dnf`, or `zipper`).

To see changes to the build history using a web interface, follow the instruction in the `README` file here. <http://git.yoctoproject.org/cgit/cgit.cgi/buildhistory-web/>.

Here is a sample screenshot of the interface:



7.29. Performing Automated Runtime Testing¶

The OpenEmbedded build system makes available a series of automated tests for images to verify runtime functionality. You can run these tests on either QEMU or actual target hardware. Tests are written in Python making use of the `unittest` module, and the majority of them run commands on the target system over SSH. This section describes how you set up the environment to use these tests, run available tests, and write and add your own tests.

For information on the test and QA infrastructure available within the Yocto Project, see the "[Testing and Quality Assurance](#)" section in the Yocto Project Reference Manual.

7.29.1. Enabling Tests¶

Depending on whether you are planning to run tests using QEMU or on the hardware, you have to take different steps to enable the tests. See the following subsections for information on how to enable both types of tests.

7.29.1.1. Enabling Runtime Tests on QEMU¶

In order to run tests, you need to do the following:

- **Set up to avoid interaction with `sudo` for networking:** To accomplish this, you must do one of the following:
 - Add `NOPASSWD` for your user in `/etc/sudoers` either for all commands or just for `runqemu-ifup`. You must provide the full path as that can change if you are using multiple clones of the source repository.

Note

On some distributions, you also need to comment out "Defaults requiretty" in `/etc/sudoers`.

- Manually configure a tap interface for your system.
- Run as root the script in `scripts/runqemu-gen-tapdevs`, which should generate a list of tap devices. This is the option typically chosen for Autobuilder-type environments.

Notes

- Be sure to use an absolute path when calling this script with `sudo`.
- The package recipe `qemu-helper-native` is required to run this script. Build the package using the following command:

```
$ bitbake qemu-helper-native
```

- **Set the `DISPLAY` variable:** You need to set this variable so that you have an X server available (e.g. start `vncserver` for a headless machine).
- **Be sure your host's firewall accepts incoming connections from `192.168.7.0/24`:** Some of the tests (in particular DNF tests) start an HTTP server on a random high number port, which is used to serve files to the target. The DNF module serves `${WORKDIR}/oe-rootfs-repo` so it can run DNF channel commands. That means your host's firewall must accept incoming connections from `192.168.7.0/24`, which is the default IP range used for tap devices by `runqemu`.
- **Be sure your host has the correct packages installed:** Depending your host's distribution, you need to have the following packages installed:
 - Ubuntu and Debian: `sysstat` and `iproute2`
 - OpenSUSE: `sysstat` and `iproute2`
 - Fedora: `sysstat` and `iproute`
 - CentOS: `sysstat` and `iproute`

Once you start running the tests, the following happens:

1. A copy of the root filesystem is written to `${WORKDIR}/testimage`.
2. The image is booted under QEMU using the standard `runqemu` script.
3. A default timeout of 500 seconds occurs to allow for the boot process to reach the login prompt. You can change the timeout period by setting `TEST_QEMUBOOT_TIMEOUT` in the `local.conf` file.
4. Once the boot process is reached and the login prompt appears, the tests run. The full boot log is written to `${WORKDIR}/testimage/qemu_boot_log`.
5. Each test module loads in the order found in `TEST_SUITES`. You can find the full output of the commands run over SSH in `${WORKDIR}/testimage/ssh_target_log`.
6. If no failures occur, the task running the tests ends successfully. You can find the output from the `unitttest` in the task log at `${WORKDIR}/temp/log.do_testimage`.

7.29.1.2. Enabling Runtime Tests on Hardware¶

The OpenEmbedded build system can run tests on real hardware, and for certain devices it can also deploy the image to be tested onto the device beforehand.

For automated deployment, a "master image" is installed onto the hardware once as part of setup. Then, each time tests are to be run, the following occurs:

1. The master image is booted into and used to write the image to be tested to a second partition.
2. The device is then rebooted using an external script that you need to provide.
3. The device boots into the image to be tested.

When running tests (independent of whether the image has been deployed automatically or not), the device is expected to be connected to a network on a pre-determined IP address. You can either use static IP addresses written into the image, or set the image to use DHCP and have your DHCP server on the test network assign a known IP address based on the MAC address of the device.

In order to run tests on hardware, you need to set `TEST_TARGET` to an appropriate value. For QEMU, you do not have to change anything, the default value is "QemuTarget". For running tests on hardware, the following options exist:

- **"SimpleRemoteTarget"**: Choose "SimpleRemoteTarget" if you are going to run tests on a target system that is already running the image to be tested and is available on the network. You can use "SimpleRemoteTarget" in conjunction with either real hardware or an image running within a separately started QEMU or any other virtual machine manager.
- **"SystemdbootTarget"**: Choose "SystemdbootTarget" if your hardware is an EFI-based machine with `systemd-boot` as bootloader and `core-image-testmaster` (or something similar) is installed. Also, your hardware under test must be in a DHCP-enabled network that gives it the same IP address for each reboot.

If you choose "SystemdbootTarget", there are additional requirements and considerations. See the ["Selecting SystemdbootTarget"](#) section, which follows, for more information.

- **"BeagleBoneTarget"**: Choose "BeagleBoneTarget" if you are deploying images and running tests on the BeagleBone "Black" or original "White" hardware. For information on how to use these tests, see the comments at the top of the `BeagleBoneTarget meta-yocto-bsp/lib/oeqa/controllers/beaglebonetarget.py` file.
- **"EdgeRouterTarget"**: Choose "EdgeRouterTarget" if you are deploying images and running tests on the Ubiquiti Networks EdgeRouter Lite. For information on how to use these tests, see the comments at the top of the `EdgeRouterTarget meta-yocto-bsp/lib/oeqa/controllers/edgeroutertarget.py` file.
- **"GrubTarget"**: Choose the "supports deploying images and running tests on any generic PC that boots using GRUB. For information on how to use these tests, see the comments at the top of the `GrubTarget meta-yocto-bsp/lib/oeqa/controllers/grubtarget.py` file.
- **"your-target"**: Create your own custom target if you want to run tests when you are deploying images and running tests on a custom machine within your BSP layer. To do this, you need to add a Python unit that defines the target class under `lib/oeqa/controllers/` within your layer. You must also provide an empty `__init__.py`. For examples, see files in `meta-yocto-bsp/lib/oeqa/controllers/`.

7.29.1.3. Selecting SystemdbootTarget¶

If you did not set `TEST_TARGET` to "SystemdbootTarget", then you do not need any information in this section. You can skip down to the ["Running Tests"](#) section.

If you did set `TEST_TARGET` to "SystemdbootTarget", you also need to perform a one-time setup of your master image by doing the following:

1. **Set *EFI_PROVIDER***: Be sure that `EFI_PROVIDER` is as follows:

```
EFI_PROVIDER = "systemd-boot"
```

2. **Build the master image**: Build the `core-image-testmaster` image. The `core-image-testmaster` recipe is provided as an example for a "master" image and you can customize the image recipe as you would any other recipe.

Here are the image recipe requirements:

- Inherits `core-image` so that kernel modules are installed.
- Installs normal linux utilities not busybox ones (e.g. `bash`, `coreutils`, `tar`, `gzip`, and `kmod`).
- Uses a custom Initial RAM Disk (`initramfs`) image with a custom installer. A normal image that you can install usually creates a single `rootfs` partition. This image uses another installer that creates a specific partition layout. Not all Board Support Packages (BSPs) can use an installer. For such cases, you need to manually create the following partition layout on the target:
 - First partition mounted under `/boot`, labeled "boot".
 - The main `rootfs` partition where this image gets installed, which is mounted under `/`.
 - Another partition labeled "testrootfs" where test images get deployed.

3. **Install image**: Install the image that you just built on the target system.

The final thing you need to do when setting `TEST_TARGET` to "SystemdbootTarget" is to set up the test image:

1. **Set up your *local.conf* file**: Make sure you have the following statements in your `local.conf` file:

```
IMAGE_FSTYPES += "tar.gz"
INHERIT += "testimage"
TEST_TARGET = "SystemdbootTarget"
TEST_TARGET_IP = "192.168.2.3"
```

2. **Build your test image**: Use BitBake to build the image:

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

7.29.1.4. Power Control¶

For most hardware targets other than SimpleRemoteTarget, you can control power:

- You can use `TEST_POWERCONTROL_CMD` together with `TEST_POWERCONTROL_EXTRA_ARGS` as a command that runs on the host and does power cycling. The test code passes one argument to that command: `off`, `on` or `cycle` (`off` then `on`). Here is an example that could appear in your `local.conf` file:

```
TEST_POWERCONTROL_CMD = "powercontrol.exp test 10.11.12.1 nuc1"
```

In this example, the expect script does the following:

```
ssh test@10.11.12.1 "pyctl nuc1 arg"
```


It then runs a Python script that controls power for a label called `nuc1`.

Note

You need to customize `TEST_POWERCONTROL_CMD` and `TEST_POWERCONTROL_EXTRA_ARGS` for your own setup. The one requirement is that it accepts "on", "off", and "cycle" as the last argument.

- When no command is defined, it connects to the device over SSH and uses the classic reboot command to reboot the device. Classic reboot is fine as long as the machine actually reboots (i.e. the SSH test has not failed). It is useful for scenarios where you have a simple setup, typically with a single board, and where some manual interaction is okay from time to time.

If you have no hardware to automatically perform power control but still wish to experiment with automated hardware testing, you can use the `dialog-power-control` script that shows a dialog prompting you to perform the required power action. This script requires either `KDialog` or `Zenity` to be installed. To use this script, set the `TEST_POWERCONTROL_CMD` variable as follows:

```
TEST_POWERCONTROL_CMD = "${COREBASE}/scripts/contrib/dialog-power-control"
```

7.29.1.5. Serial Console Connection¶

For test target classes requiring a serial console to interact with the bootloader (e.g. `BeagleBoneTarget`, `EdgeRouterTarget`, and `GrubTarget`), you need to specify a command to use to connect to the serial console of the target machine by using the `TEST_SERIALCONTROL_CMD` variable and optionally the `TEST_SERIALCONTROL_EXTRA_ARGS` variable.

These cases could be a serial terminal program if the machine is connected to a local serial port, or a `telnet` or `ssh` command connecting to a remote console server. Regardless of the case, the command simply needs to connect to the serial console and forward that connection to standard input and output as any normal terminal program does. For example, to use the `picocom` terminal program on serial device `/dev/ttyUSB0` at 115200bps, you would set the variable as follows:

```
TEST_SERIALCONTROL_CMD = "picocom /dev/ttyUSB0 -b 115200"
```

For local devices where the serial port device disappears when the device reboots, an additional "serdevtry" wrapper script is provided. To use this wrapper, simply prefix the terminal command with `${COREBASE}/scripts/contrib/serdevtry`:

```
TEST_SERIALCONTROL_CMD = "${COREBASE}/scripts/contrib/serdevtry picocom -b 115200 /dev/ttyUSB0"
```

7.29.2. Running Tests¶

You can start the tests automatically or manually:

- Automatically running tests:** To run the tests automatically after the OpenEmbedded build system successfully creates an image, first set the `TESTIMAGE_AUTO` variable to "1" in your `local.conf` file in the [Build Directory](#):

```
TESTIMAGE_AUTO = "1"
```

Next, build your image. If the image successfully builds, the tests run:

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

- Manually running tests:** To manually run the tests, first globally inherit the `testimage` class by editing your `local.conf` file:

```
INHERIT += "testimage"
```

Next, use BitBake to run the tests:

```
bitbake -c testimage image
```

All test files reside in `meta/lib/oeqa/runtime` in the [Source Directory](#). A test name maps directly to a Python module. Each test module may contain a number of individual tests. Tests are usually grouped together by the area tested (e.g. tests for `systemd` reside in `meta/lib/oeqa/runtime/systemd.py`).

You can add tests to any layer provided you place them in the proper area and you extend `BBPATH` in the `local.conf` file as normal. Be sure that tests reside in `Layer/lib/oeqa/runtime`.

Note

Be sure that module names do not collide with module names used in the default set of test modules in `meta/lib/oeqa/runtime`.

You can change the set of tests run by appending or overriding `TEST_SUITES` variable in `local.conf`. Each name in `TEST_SUITES` represents a required test for the image. Test modules named within `TEST_SUITES` cannot be skipped even if a test is not suitable for an image (e.g. running the RPM tests on an image without `rpm`). Appending "auto" to `TEST_SUITES` causes the build system to try to run all tests that are suitable for the image (i.e. each test module may elect to skip itself).

The order you list tests in `TEST_SUITES` is important and influences test dependencies. Consequently, tests that depend on other tests should be added after the test on which they depend. For example, since the `ssh` test depends on the `ping` test, "ssh" needs to come after "ping" in the list. The test class provides no re-ordering or dependency handling.

Note

Each module can have multiple classes with multiple test methods. And, Python `unittest` rules apply.

Here are some things to keep in mind when running tests:

- The default tests for the image are defined as:

```
DEFAULT_TEST_SUITES_pn-image = "ping ssh df connman syslog xorg scp vnc date rpm dnf dmesg"
```

- Add your own test to the list of the by using the following:

```
TEST_SUITES_append = " mytest"
```

- Run a specific list of tests as follows:

```
TEST_SUITES = "test1 test2 test3"
```

Remember, order is important. Be sure to place a test that is dependent on another test later in the order.

7.29.3. Exporting Tests¶

You can export tests so that they can run independently of the build system. Exporting tests is required if you want to be able to hand the test execution off to a scheduler. You can only export tests that are defined in `TEST_SUITES`.

If your image is already built, make sure the following are set in your `local.conf` file:

```
INHERIT += "testexport"
TEST_TARGET_IP = "IP-address-for-the-test-target"
TEST_SERVER_IP = "IP-address-for-the-test-server"
```

You can then export the tests with the following BitBake command form:

```
$ bitbake image -c testexport
```

Exporting the tests places them in the [Build Directory](#) in `tmp/testexport/image`, which is controlled by the `TEST_EXPORT_DIR` variable.

You can now run the tests outside of the build environment:

```
$ cd tmp/testexport/image
$ ./runexported.py testdata.json
```

Here is a complete example that shows IP addresses and uses the `core-image-sato` image:

```
INHERIT += "testexport"
TEST_TARGET_IP = "192.168.7.2"
TEST_SERVER_IP = "192.168.7.1"
```

Use BitBake to export the tests:

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

Run the tests outside of the build environment using the following:

```
$ cd tmp/testexport/core-image-sato
$ ./runexported.py testdata.json
```

7.29.4. Writing New Tests¶

As mentioned previously, all new test files need to be in the proper place for the build system to find them. New tests for additional functionality outside of the core should be added to the layer that adds the functionality, in `layer/lib/oeqa/runtime` (as long as `BBPATH` is extended in the layer's `layer.conf` file as normal). Just remember the following:

- Filenames need to map directly to test (module) names.
- Do not use module names that collide with existing core tests.
- Minimally, an empty `__init__.py` file must exist in the runtime directory.

To create a new test, start by copying an existing module (e.g. `syslog.py` or `gcc.py` are good ones to use). Test modules can use code from `meta/lib/oeqa/utlis`, which are helper classes.

Note

Structure shell commands such that you rely on them and they return a single code for success. Be aware that sometimes you will need to parse the output. See the `df.py` and `date.py` modules for examples.

You will notice that all test classes inherit `oeRuntimeTest`, which is found in `meta/lib/oetest.py`. This base class offers some helper attributes, which are described in the following sections:

7.29.4.1. Class Methods¶

Class methods are as follows:

- **hasPackage (pkg)**: Returns "True" if `pkg` is in the installed package list of the image, which is based on the manifest file that is generated during the `do_rootfs` task.
- **hasFeature (feature)**: Returns "True" if the feature is in `IMAGE_FEATURES` or `DISTRO_FEATURES`.

7.29.4.2. Class Attributes¶

Class attributes are as follows:

- **pscmd**: Equals "ps -ef" if `procps` is installed in the image. Otherwise, `pscmd` equals "ps" (busybox).
- **tc**: The called test context, which gives access to the following attributes:
 - **d**: The BitBake datastore, which allows you to use stuff such as `oeRuntimeTest.tc.d.getVar("VIRTUAL-RUNTIME_init_manager")`.
 - **testslint and testsrequired**: Used internally. The tests do not need these.
 - **filesdir**: The absolute path to `meta/lib/oeqa/runtime/files`, which contains helper files for tests meant for copying on the target such as small files written in C for compilation.
 - **target**: The target controller object used to deploy and start an image on a particular target (e.g. `QemuTarget`, `SimpleRemote`, and `SystemdbootTarget`). Tests usually use the following:
 - **ip**: The target's IP address.
 - **server_ip**: The host's IP address, which is usually used by the DNF test suite.
 - **run(cmd, timeout=None)**: The single, most used method. This command is a wrapper for: `ssh root@host "cmd"`. The command returns a tuple: (status, output), which are what their names imply - the return code of "cmd" and whatever output it produces. The optional timeout argument represents the number of seconds the test should wait for "cmd" to return. If the argument is "None", the test uses the default instance's timeout period, which is 300 seconds. If the argument is "0", the test runs until the command returns.
 - **copy_to(localpath, remotepath)**: `scp localpath root@ip:remotepath`.
 - **copy_from(remotepath, localpath)**: `scp root@host:remotepath localpath`.

7.29.4.3. Instance Attributes¶

A single instance attribute exists, which is `target`. The `target` instance attribute is identical to the class attribute of the same name, which is described in the previous section. This attribute exists as both an instance and class attribute so tests can use `self.target.run(cmd)` in instance methods instead of `oeRuntimeTest.tc.target.run(cmd)`.

7.29.5. Installing Packages in the DUT Without the Package Manager¶

When a test requires a package built by BitBake, it is possible to install that package. Installing the package does not require a package manager be installed in the device under test (DUT). It does, however, require an SSH connection and the target must be using the `sshcontrol` class.

Note

This method uses `scp` to copy files from the host to the target, which causes permissions and special attributes to be lost.

A JSON file is used to define the packages needed by a test. This file must be in the same path as the file used to define the tests. Furthermore, the filename must map directly to the test module name with a `.json` extension.

The JSON file must include an object with the test name as keys of an object or an array. This object (or array of objects) uses the following data:

- "pkg" - A mandatory string that is the name of the package to be installed.
- "rm" - An optional boolean, which defaults to "false", that specifies to remove the package after the test.
- "extract" - An optional boolean, which defaults to "false", that specifies if the package must be extracted from the package format. When set to "true", the package is not automatically installed into the DUT.

Following is an example JSON file that handles test "foo" installing package "bar" and test "foobar" installing packages "foo" and "bar". Once the test is complete, the packages are removed from the DUT.

```
{
  "foo": {
    "pkg": "bar"
  },
  "foobar": [
    {
      "pkg": "foo",
      "rm": true
    },
    {
      "pkg": "bar",
      "rm": true
    }
  ]
}
```

}

7.30. Debugging Tools and Techniques¶

The exact method for debugging build failures depends on the nature of the problem and on the system's area from which the bug originates. Standard debugging practices such as comparison against the last known working version with examination of the changes and the re-application of steps to identify the one causing the problem are valid for the Yocto Project just as they are for any other system. Even though it is impossible to detail every possible potential failure, this section provides some general tips to aid in debugging given a variety of situations.

Tip

A useful feature for debugging is the error reporting tool. Configuring the Yocto Project to use this tool causes the OpenEmbedded build system to produce error reporting commands as part of the console output. You can enter the commands after the build completes to log error information into a common database, that can help you figure out what might be going wrong. For information on how to enable and use this feature, see the ["Using the Error Reporting Tool"](#) section.

The following list shows the debugging topics in the remainder of this section:

- ["Viewing Logs from Failed Tasks"](#) describes how to find and view logs from tasks that failed during the build process.
- ["Viewing Variable Values"](#) describes how to use the BitBake `-e` option to examine variable values after a recipe has been parsed.
- ["Viewing Package Information with oe-pkgdata-util"](#) describes how to use the `oe-pkgdata-util` utility to query `PKGDATA_DIR` and display package-related information for built packages.
- ["Viewing Dependencies Between Recipes and Tasks"](#) describes how to use the BitBake `-g` option to display recipe dependency information used during the build.
- ["Viewing Task Variable Dependencies"](#) describes how to use the `bitbake-dumpsig` command in conjunction with key subdirectories in the [Build Directory](#) to determine variable dependencies.
- ["Running Specific Tasks"](#) describes how to use several BitBake options (e.g. `-C`, `-C`, and `-f`) to run specific tasks in the build chain. It can be useful to run tasks "out-of-order" when trying isolate build issues.
- ["General BitBake Problems"](#) describes how to use BitBake's `-D` debug output option to reveal more about what BitBake is doing during the build.
- ["Building with No Dependencies"](#) describes how to use the BitBake `-b` option to build a recipe while ignoring dependencies.
- ["Recipe Logging Mechanisms"](#) describes how to use the many recipe logging functions to produce debugging output and report errors and warnings.
- ["Debugging Parallel Make Races"](#) describes how to debug situations where the build consists of several parts that are run simultaneously and when the output or result of one part is not ready for use with a different part of the build that depends on that output.
- ["Debugging With the GNU Project Debugger \(GDB\) Remotely"](#) describes how to use GDB to allow you to examine running programs, which can help you fix problems.
- ["Debugging with the GNU Project Debugger \(GDB\) on the Target"](#) describes how to use GDB directly on target hardware for debugging.
- ["Other Debugging Tips"](#) describes miscellaneous debugging tips that can be useful.

7.30.1. Viewing Logs from Failed Tasks¶

You can find the log for a task in the file `${WORKDIR}/temp/log.do_taskname`. For example, the log for the `do_compile` task of the QEMU minimal image for the x86 machine (`qemux86`) might be in `tmp/work/qemux86-poky-linux/core-image-minimal/1.0-r0/temp/log.do_compile`. To see the commands [BitBake](#) ran to generate a log, look at the corresponding `run.do_taskname` file in the same directory.

`log.do_taskname` and `run.do_taskname` are actually symbolic links to `log.do_taskname.pid` and `log.run_taskname.pid`, where `pid` is the PID the task had when it ran. The symlinks always point to the files corresponding to the most recent run.

7.30.2. Viewing Variable Values¶

Sometimes you need to know the value of a variable as a result of BitBake's parsing step. This could be because some unexpected behavior occurred in your project. Perhaps an attempt to [modify a variable](#) did not work out as expected.

BitBake's `-e` option is used to display variable values after parsing. The following command displays the variable values after the configuration files (i.e. `local.conf`, `bblayers.conf`, `bitbake.conf` and so forth) have been parsed:

```
$ bitbake -e
```

The following command displays variable values after a specific recipe has been parsed. The variables include those from the configuration as well:

```
$ bitbake -e recipename
```

Note

Each recipe has its own private set of variables (datastore). Internally, after parsing the configuration, a copy of the resulting datastore is made prior to parsing each recipe. This copying implies that variables set in one recipe will not be visible to other recipes.

Likewise, each task within a recipe gets a private datastore based on the recipe datastore, which means that variables set within one task will not be visible to other tasks.

In the output of `bitbake -e`, each variable is preceded by a description of how the variable got its value, including temporary values that were later overridden. This description also includes variable flags (varflags) set on the variable. The output can be very helpful during debugging.

Variables that are exported to the environment are preceded by `export` in the output of `bitbake -e`. See the following example:

```
export CC="i586-poky-linux-gcc -m32 -march=i586 --sysroot=/home/ulf/poky/build/tmp/sysroots/qemux86"
```

In addition to variable values, the output of the `bitbake -e` and `bitbake -e recipe` commands includes the following information:

- The output starts with a tree listing all configuration files and classes included globally, recursively listing the files they include or inherit in turn. Much of the behavior of the OpenEmbedded build system (including the behavior of the [normal recipe build tasks](#)) is implemented in the `base` class and the classes it inherits, rather than being built into BitBake itself.
- After the variable values, all functions appear in the output. For shell functions, variables referenced within the function body are expanded. If a function has been modified using overrides or using override-style operators like `_append` and `_prepend`, then the final assembled function body appears in the output.

7.30.3. Viewing Package Information with `oe-pkgdata-util`

You can use the `oe-pkgdata-util` command-line utility to query `PKGDATA_DIR` and display various package-related information. When you use the utility, you must use it to view information on packages that have already been built.

Following are a few of the available `oe-pkgdata-util` subcommands.

Note

You can use the standard `*` and `?` globbing wildcards as part of package names and paths.

- `oe-pkgdata-util list-pkgs [pattern]`: Lists all packages that have been built, optionally limiting the match to packages that match *pattern*.
- `oe-pkgdata-util list-pkg-files package ...`: Lists the files and directories contained in the given packages.

Note

A different way to view the contents of a package is to look at the `${WORKDIR}/packages-split` directory of the recipe that generates the package. This directory is created by the `do_package` task and has one subdirectory for each package the recipe generates, which contains the files stored in that package.

If you want to inspect the `${WORKDIR}/packages-split` directory, make sure that `rm_work` is not enabled when you build the recipe.

- `oe-pkgdata-util find-path path ...`: Lists the names of the packages that contain the given paths. For example, the following tells us that `/usr/share/man/man1/make.1` is contained in the `make-doc` package:


```
$ oe-pkgdata-util find-path /usr/share/man/man1/make.1
make-doc: /usr/share/man/man1/make.1
```
- `oe-pkgdata-util lookup-recipe package ...`: Lists the name of the recipes that produce the given packages.

For more information on the `oe-pkgdata-util` command, use the help facility:

```
$ oe-pkgdata-util --help
$ oe-pkgdata-util subcommand --help
```

7.30.4. Viewing Dependencies Between Recipes and Tasks

Sometimes it can be hard to see why BitBake wants to build other recipes before the one you have specified. Dependency information can help you understand why a recipe is built.

To generate dependency information for a recipe, run the following command:

```
$ bitbake -g recipename
```

This command writes the following files in the current directory:

- `pn-buildlist`: A list of recipes/targets involved in building *recipe*. "Involved" here means that at least one task from the recipe needs to run when building *recipe* from scratch. Targets that are in `ASSUME_PROVIDED` are not listed.
- `task-depends.dot`: A graph showing dependencies between tasks.

The graphs are in `DOT` format and can be converted to images (e.g. using the `dot` tool from [Graphviz](#)).

Notes

- DOT files use a plain text format. The graphs generated using the `bitbake -g` command are often so large as to be difficult to read without special pruning (e.g. with BitBake's `-I` option) and processing. Despite the form and size of the graphs, the corresponding `.dot` files can still be possible to read and provide useful information.

As an example, the `task-depends.dot` file contains lines such as the following:

```
"libxslt.do_configure" -> "libxml2.do_populate_sysroot"
```

The above example line reveals that the `do_configure` task in `libxslt` depends on the `do_populate_sysroot` task in `libxml2`, which is a normal `DEPENDS` dependency between the two recipes.

- For an example of how `.dot` files can be processed, see the `scripts/contrib/graph-tool` Python script, which finds and displays paths between graph nodes.

You can use a different method to view dependency information by using the following command:

```
$ bitbake -g -u taskexp recipe
```

This command displays a GUI window from which you can view build-time and runtime dependencies for the recipes involved in building *recipe*.

7.30.5. Viewing Task Variable Dependencies¶

As mentioned in the "[Checksums \(Signatures\)](#)" section of the BitBake User Manual, BitBake tries to automatically determine what variables a task depends on so that it can rerun the task if any values of the variables change. This determination is usually reliable. However, if you do things like construct variable names at runtime, then you might have to manually declare dependencies on those variables using `vardeps` as described in the "[Variable Flags](#)" section of the BitBake User Manual.

If you are unsure whether a variable dependency is being picked up automatically for a given task, you can list the variable dependencies BitBake has determined by doing the following:

1. Build the recipe containing the task:

```
$ bitbake recipe
```

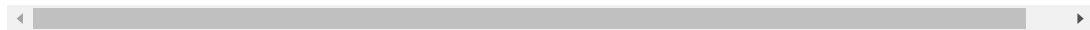
2. Inside the `STAMPS_DIR` directory, find the signature data (`sigdata`) file that corresponds to the task. The `sigdata` files contain a pickled Python database of all the metadata that went into creating the input checksum for the task. As an example, for the `do_fetch` task of the `db` recipe, the `sigdata` file might be found in the following location:

```
${BUILDDIR}/tmp/stamps/i586-poky-linux/db/6.0.30-r1.do_fetch.sigdata.7c048c18222b16ff0bcee200ef648b1
```

For tasks that are accelerated through the shared state (`sstate`) cache, an additional `siginfo` file is written into `SSTATE_DIR` along with the cached task output. The `siginfo` files contain exactly the same information as `sigdata` files.

3. Run `bitbake-dumpsig` on the `sigdata` or `siginfo` file. Here is an example:

```
$ bitbake-dumpsig ${BUILDDIR}/tmp/stamps/i586-poky-linux/db/6.0.30-r1.do_fetch.sigdata.7c048c18222b16ff0bcee200ef648b1
```



In the output of the above command, you will find a line like the following, which lists all the (inferred) variable dependencies for the task. This list also includes indirect dependencies from variables depending on other variables, recursively.

```
Task dependencies: ['PV', 'SRCREV', 'SRC_URI', 'SRC_URI[md5sum]', 'SRC_URI[sha256sum]', 'base_do_fetch']
```

Note

Functions (e.g. `base_do_fetch`) also count as variable dependencies. These functions in turn depend on the variables they reference.

The output of `bitbake-dumpsig` also includes the value each variable had, a list of dependencies for each variable, and `BB_HASHBASE_WHITELIST` information.

There is also a `bitbake-diffsigs` command for comparing two `siginfo` or `sigdata` files. This command can be helpful when trying to figure out what changed between two versions of a task. If you call `bitbake-diffsigs` with just one file, the command behaves like `bitbake-dumpsig`.

You can also use BitBake to dump out the signature construction information without executing tasks by using either of the following BitBake command-line options:

```
--dump-signatures=SIGNATURE_HANDLER
-S SIGNATURE_HANDLER
```

Note

Two common values for `SIGNATURE_HANDLER` are "none" and "printdiff", which dump only the signature or compare the dumped signature with the cached one, respectively.

Using BitBake with either of these options causes BitBake to dump out `sigdata` files in the `stamps` directory for every task it would have executed instead of building the specified target package.

7.30.6. Viewing Metadata Used to Create the Input Signature of a Shared State Task¶

Seeing what metadata went into creating the input signature of a shared state (`sstate`) task can be a useful debugging aid. This information is available in signature information (`siginfo`) files in `SSTATE_DIR`. For information on how to view and interpret information in `siginfo` files, see the "[Viewing Task Variable Dependencies](#)" section.

For conceptual information on shared state, see the "[Shared State](#)" section in the Yocto Project Overview and Concepts Manual.

7.30.7. Invalidating Shared State to Force a Task to Run¶

The OpenEmbedded build system uses [checksums](#) and [shared state](#) cache to avoid unnecessarily rebuilding tasks. Collectively, this scheme is known as "shared state code."

As with all schemes, this one has some drawbacks. It is possible that you could make implicit changes to your code that the checksum calculations do not take into account. These implicit changes affect a task's output but do not trigger the shared state code into rebuilding a recipe. Consider an example during which a tool changes its output. Assume that the output of `rpmdeps` changes. The result of the change should be that all the `package` and `package_write_rpm` shared state cache items become invalid. However, because the change to the output is external to the code and therefore implicit, the associated shared state cache items do not become invalidated. In this case, the build process uses the cached items rather than running the task again. Obviously, these types of implicit changes can cause problems.

To avoid these problems during the build, you need to understand the effects of any changes you make. Realize that changes you make directly to a function are automatically factored into the checksum calculation. Thus, these explicit changes invalidate the associated area of shared state cache. However, you need to be aware of any implicit changes that are not obvious changes to the code and could affect the output of a given task.

When you identify an implicit change, you can easily take steps to invalidate the cache and force the tasks to run. The steps you can take are as simple as changing a function's comments in the source code. For example, to invalidate package shared state files, change the comment statements of `do_package` or the comments of one of the functions it calls. Even though the change is purely cosmetic, it causes the checksum to be recalculated and forces the build system to run the task again.

Note

For an example of a commit that makes a cosmetic change to invalidate shared state, see this [commit](#).

7.30.8. Running Specific Tasks¶

Any given recipe consists of a set of tasks. The standard BitBake behavior in most cases is: `do_fetch`, `do_unpack`, `do_patch`, `do_configure`, `do_compile`, `do_install`, `do_package`, `do_package_write_*`, and `do_build`. The default task is `do_build` and any tasks on which it depends build first. Some tasks, such as `do_devshell`, are not part of the default build chain. If you wish to run a task that is not part of the default build chain, you can use the `-c` option in BitBake. Here is an example:

```
$ bitbake matchbox-desktop -c devshell
```

The `-C` option respects task dependencies, which means that all other tasks (including tasks from other recipes) that the specified task depends on will be run before the task. Even when you manually specify a task to run with `-C`, BitBake will only run the task if it considers it "out of date". See the "[Stamp Files and the Rerunning of Tasks](#)" section in the Yocto Project Overview and Concepts Manual for how BitBake determines whether a task is "out of date".

If you want to force an up-to-date task to be rerun (e.g. because you made manual modifications to the recipe's `WORKDIR` that you want to try out), then you can use the `-f` option.

Note

The reason `-f` is never required when running the `do_devshell` task is because the `[nostamp]` variable flag is already set for the task.

The following example shows one way you can use the `-f` option:

```
$ bitbake matchbox-desktop
.
.
make some changes to the source code in the work directory
.
.
$ bitbake matchbox-desktop -c compile -f
```



```
$ bitbake matchbox-desktop
```

This sequence first builds and then recompiles `matchbox-desktop`. The last command reruns all tasks (basically the packaging tasks) after the compile. BitBake recognizes that the `do_compile` task was rerun and therefore understands that the other tasks also need to be run again.

Another, shorter way to rerun a task and all normal recipe build tasks that depend on it is to use the `-C` option.

Note

This option is upper-cased and is separate from the `-C` option, which is lower-cased.

Using this option invalidates the given task and then runs the `do_build` task, which is the default task if no task is given, and the tasks on which it depends. You could replace the final two commands in the previous example with the following single command:

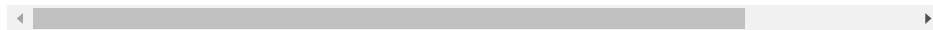
```
$ bitbake matchbox-desktop -C compile
```

Internally, the `-f` and `-C` options work by tainting (modifying) the input checksum of the specified task. This tainting indirectly causes the task and its dependent tasks to be rerun through the normal task dependency mechanisms.

Note

BitBake explicitly keeps track of which tasks have been tainted in this fashion, and will print warnings such as the following for builds involving such tasks:

```
WARNING: /home/ulf/poky/meta/recipes-sato/matchbox-desktop/matchbox-desktop_2.1.bb.do_compile is t
```



The purpose of the warning is to let you know that the work directory and build output might not be in the clean state they would be in for a "normal" build, depending on what actions you took. To get rid of such warnings, you can remove the work directory and rebuild the recipe, as follows:

```
$ bitbake matchbox-desktop -c clean
$ bitbake matchbox-desktop
```

You can view a list of tasks in a given package by running the `do_listtasks` task as follows:

```
$ bitbake matchbox-desktop -c listtasks
```

The results appear as output to the console and are also in the file `${WORKDIR}/temp/log.do_listtasks`.

7.30.9. General BitBake Problems¶

You can see debug output from BitBake by using the `-D` option. The debug output gives more information about what BitBake is doing and the reason behind it. Each `-D` option you use increases the logging level. The most common usage is `-DDD`.

The output from `bitbake -DDD -v targetname` can reveal why BitBake chose a certain version of a package or why BitBake picked a certain provider. This command could also help you in a situation where you think BitBake did something unexpected.

7.30.10. Building with No Dependencies¶

To build a specific recipe (`.bb` file), you can use the following command form:

```
$ bitbake -b somepath/somerecipe.bb
```

This command form does not check for dependencies. Consequently, you should use it only when you know existing dependencies have been met.

Note

You can also specify fragments of the filename. In this case, BitBake checks for a unique match.

7.30.11. Recipe Logging Mechanisms¶

The Yocto Project provides several logging functions for producing debugging output and reporting errors and warnings. For Python functions, the following logging functions exist. All of these functions log to `${T}/log.do_task`, and can also log to standard output (stdout) with the right settings:

- `bb.plain(msg)` : Writes `msg` as is to the log while also logging to stdout.
- `bb.note(msg)` : Writes "NOTE: `msg`" to the log. Also logs to stdout if BitBake is called with `"-v"`.
- `bb.debug(level, msg)` : Writes "DEBUG: `msg`" to the log. Also logs to stdout if the log level is greater than or equal to `level`. See the `"-D"` option in the BitBake User Manual for more information.
- `bb.warn(msg)` : Writes "WARNING: `msg`" to the log while also logging to stdout.

- `bb.error(msg)` : Writes "ERROR: *msg*" to the log while also logging to standard out (stdout).

Note

Calling this function does not cause the task to fail.

- `bb.fatal(msg)` : This logging function is similar to `bb.error(msg)` but also causes the calling task to fail.

Note

`bb.fatal()` raises an exception, which means you do not need to put a "return" statement after the function.

The same logging functions are also available in shell functions, under the names `bbplain`, `bbnote`, `bbdebug`, `bbwarn`, `bberror`, and `bbfatal`. The `logging` class implements these functions. See that class in the `meta/classes` folder of the [Source Directory](#) for information.

7.30.11.1. Logging With Python¶

When creating recipes using Python and inserting code that handles build logs, keep in mind the goal is to have informative logs while keeping the console as "silent" as possible. Also, if you want status messages in the log, use the "debug" loglevel.

Following is an example written in Python. The code handles logging for a function that determines the number of tasks needed to be run. See the [do_listtasks](#) section for additional information:

```
python do_listtasks() {
    bb.debug(2, "Starting to figure out the task list")
    if noteworthy_condition:
        bb.note("There are 47 tasks to run")
    bb.debug(2, "Got to point xyz")
    if warning_trigger:
        bb.warn("Detected warning_trigger, this might be a problem later.")
    if recoverable_error:
        bb.error("Hit recoverable_error, you really need to fix this!")
    if fatal_error:
        bb.fatal("fatal_error detected, unable to print the task list")
    bb.plain("The tasks present are abc")
    bb.debug(2, "Finished figuring out the tasklist")
}
```

7.30.11.2. Logging With Bash¶

When creating recipes using Bash and inserting code that handles build logs, you have the same goals - informative with minimal console output. The syntax you use for recipes written in Bash is similar to that of recipes written in Python described in the previous section.

Following is an example written in Bash. The code logs the progress of the `do_my_function` function.

```
do_my_function() {
    bbdebug 2 "Running do_my_function"
    if [ exceptional_condition ]; then
        bbnote "Hit exceptional_condition"
    fi
    bbdebug 2 "Got to point xyz"
    if [ warning_trigger ]; then
        bbwarn "Detected warning_trigger, this might cause a problem later."
    fi
    if [ recoverable_error ]; then
        bberror "Hit recoverable_error, correcting"
    fi
    if [ fatal_error ]; then
        bbfatal "fatal_error detected"
    fi
    bbdebug 2 "Completed do_my_function"
}
```

7.30.12. Debugging Parallel Make Races¶

A parallel `make` race occurs when the build consists of several parts that are run simultaneously and a situation occurs when the output or result of one part is not ready for use with a different part of the build that depends on that output. Parallel make races are annoying and can sometimes be difficult to reproduce and fix. However, some simple tips and tricks exist that can help you debug and fix them. This section presents a real-world example of an error encountered on the Yocto Project autobuilder and the process used to fix it.

Note

If you cannot properly fix a `make` race condition, you can work around it by clearing either the `PARALLEL_MAKE` or `PARALLEL_MAKEINST` variables.

7.30.12.1. The Failure¶

For this example, assume that you are building an image that depends on the "neard" package. And, during the build, BitBake runs into problems and creates the following output.

Note

This example log file has longer lines artificially broken to make the listing easier to read.

If you examine the output or the log file, you see the failure during `make`:

```
| DEBUG: SITE files ['endian-little', 'bit-32', 'ix86-common', 'common-linux', 'common-glibc', 'i586-linux', 'common']
| DEBUG: Executing shell function do_compile
| NOTE: make -j 16
| make --no-print-directory all-am
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
0.14-r0/neard-0.14/include/types.h include/near/types.h
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
0.14-r0/neard-0.14/include/log.h include/near/log.h
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
0.14-r0/neard-0.14/include/plugin.h include/near/plugin.h
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
0.14-r0/neard-0.14/include/tag.h include/near/tag.h
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
0.14-r0/neard-0.14/include/adaptor.h include/near/adaptor.h
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
0.14-r0/neard-0.14/include/ndef.h include/near/ndef.h
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
0.14-r0/neard-0.14/include/tlv.h include/near/tlv.h
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
0.14-r0/neard-0.14/include/setting.h include/near/setting.h
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
0.14-r0/neard-0.14/include/device.h include/near/device.h
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
0.14-r0/neard-0.14/include/nfc_copy.h include/near/nfc_copy.h
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
0.14-r0/neard-0.14/include/snep.h include/near/snep.h
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
0.14-r0/neard-0.14/include/version.h include/near/version.h
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
0.14-r0/neard-0.14/include/dbus.h include/near/dbus.h
| ./src/genbuiltin nfctype1 nfctype2 nfctype3 nfctype4 p2p > src/builtin.h
| i586-poky-linux-gcc -m32 -march=i586 --sysroot=/home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/
build/build/tmp/sysroots/qemux86 -DHAVE_CONFIG_H -I. -I./include -I./src -I./gdbus -I/home/pokybuild/
yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/sysroots/qemux86/usr/include/glib-2.0
-I/home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/sysroots/qemux86/usr/
lib/glib-2.0/include -I/home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/
tmp/sysroots/qemux86/usr/include/dbus-1.0 -I/home/pokybuild/yocto-autobuilder/yocto-slave/
nightly-x86/build/build/tmp/sysroots/qemux86/usr/lib/dbus-1.0/include -I/home/pokybuild/yocto-autobuilder/
yocto-slave/nightly-x86/build/build/tmp/sysroots/qemux86/usr/include/libn13
-DNEAR_PLUGIN_BUILTIN -DPLUGININDIR="" /usr/lib/near/plugins""
-DCONFIGDIR="" /etc/neard"" -O2 -pipe -g -feliminate-unused-debug-types -c
-o tools/snep-send.o tools/snep-send.c
| In file included from tools/snep-send.c:16:0:
| tools/./src/near.h:41:23: fatal error: near/dbus.h: No such file or directory
| #include <near/dbus.h>
| ^
| compilation terminated.
| make[1]: *** [tools/snep-send.o] Error 1
| make[1]: *** Waiting for unfinished jobs....
| make: *** [all] Error 2
| ERROR: oe_runmake failed
```

7.30.12.2. Reproducing the Error¶

Because race conditions are intermittent, they do not manifest themselves every time you do the build. In fact, most times the build will complete without problems even though the potential race condition exists. Thus, once the error surfaces, you need a way to reproduce it.

In this example, compiling the "neard" package is causing the problem. So the first thing to do is build "neard" locally. Before you start the build, set the `PARALLEL_MAKE` variable in your `local.conf` file to a high number (e.g. "-j 20"). Using a high value for `PARALLEL_MAKE` increases the chances of the race condition showing up:

```
$ bitbake neard
```

Once the local build for "neard" completes, start a `devshell` build:

```
$ bitbake neard -c devshell
```

For information on how to use a `devshell`, see the "Using a Development Shell" section.

In the `devshell`, do the following:

```
$ make clean
$ make tools/snep-send.o
```

The `devshell` commands cause the failure to clearly be visible. In this case, a missing dependency exists for the "neard" Makefile target. Here is some abbreviated, sample output with the missing dependency clearly visible at the end:

```
i586-poky-linux-gcc -m32 -march=i586 --sysroot=/home/scott-lenovo/.....
.
.
.
tools/snep-send.c
In file included from tools/snep-send.c:16:0:
tools/./src/near.h:41:23: fatal error: near/dbus.h: No such file or directory
#include <near/dbus.h>
^
compilation terminated.
make: *** [tools/snep-send.o] Error 1
$
```

7.30.12.3. Creating a Patch for the Fix¶

Because there is a missing dependency for the Makefile target, you need to patch the `Makefile.am` file, which is generated from `Makefile.in`. You can use Quilt to create the patch:

```
$ quilt new parallelmake.patch
Patch patches/parallelmake.patch is now on top
$ quilt add Makefile.am
File Makefile.am added to patch patches/parallelmake.patch
```

For more information on using Quilt, see the "[Using Quilt in Your Workflow](#)" section.

At this point you need to make the edits to `Makefile.am` to add the missing dependency. For our example, you have to add the following line to the file:

```
tools/snep-send.$(OBJEXT): include/near/dbus.h
```

Once you have edited the file, use the `refresh` command to create the patch:

```
$ quilt refresh
Refreshed patch patches/parallelmake.patch
```

Once the patch file exists, you need to add it back to the originating recipe folder. Here is an example assuming a top-level [Source Directory](#) named `poky`:

```
$ cp patches/parallelmake.patch poky/meta/recipes-connectivity/neard/neard
```

The final thing you need to do to implement the fix in the build is to update the "neard" recipe (i.e. `neard-0.14.bb`) so that the `SRC_URI` statement includes the patch file. The recipe file is in the folder above the patch. Here is what the edited `SRC_URI` statement would look like:

```
SRC_URI = "${KERNELORG_MIRROR}/linux/network/nfc/${BPN}-${PV}.tar.xz \
    file://neard.in \
    file://neard.service.in \
    file://parallelmake.patch \
"
```

With the patch complete and moved to the correct folder and the `SRC_URI` statement updated, you can exit the `devshell`:

```
$ exit
```

7.30.12.4. Testing the Build¶

With everything in place, you can get back to trying the build again locally:

```
$ bitbake neard
```

This build should succeed.

Now you can open up a `devshell` again and repeat the clean and make operations as follows:

```
$ bitbake neard -c devshell
$ make clean
$ make tools/snep-send.o
```

The build should work without issue.

As with all solved problems, if they originated upstream, you need to submit the fix for the recipe in OE-Core and upstream so that the problem is taken care of at its source. See the "[Submitting a Change to the Yocto Project](#)" section for more information.

7.30.13. Debugging With the GNU Project Debugger (GDB) Remotely¶

GDB allows you to examine running programs, which in turn helps you to understand and fix problems. It also allows you to perform post-mortem style analysis of program crashes. GDB is available as a package within the Yocto Project and is installed in SDK images by default. See the "[Images](#)" chapter in the Yocto Project Reference Manual for a description of these images. You can find information on GDB at <http://sourceware.org/gdb/>.

Tip

For best results, install debug (`-dbg`) packages for the applications you are going to debug. Doing so makes extra debug symbols available that give you more meaningful output.

Sometimes, due to memory or disk space constraints, it is not possible to use GDB directly on the remote target to debug applications. These constraints arise because GDB needs to load the debugging information and the binaries of the process being debugged. Additionally, GDB needs to perform many computations to locate information such as function names, variable names and values, stack traces and so forth - even before starting the debugging process. These extra computations place more load on the target system and can alter the characteristics of the program being debugged.

To help get past the previously mentioned constraints, you can use `gdbserver`, which runs on the remote target and does not load any debugging information from the debugged process. Instead, a GDB instance processes the debugging information that is run on a remote computer - the host GDB. The host GDB then sends control commands to `gdbserver` to make it stop or start the debugged program, as well as read or write memory regions of that debugged program. All the debugging information loaded and processed as well as all the heavy debugging is done by the host GDB. Offloading these processes gives the `gdbserver` running on the target a chance to remain small and fast.

Because the host GDB is responsible for loading the debugging information and for doing the necessary processing to make actual debugging happen, you have to make sure the host can access the unstripped binaries complete with their debugging information and also be sure the target is compiled with no optimizations. The host GDB must also have local access to all the libraries used by the debugged program. Because `gdbserver` does not need any local debugging information, the binaries on the remote target can remain stripped. However, the binaries must also be compiled without optimization so they match the host's binaries.

To remain consistent with GDB documentation and terminology, the binary being debugged on the remote target machine is referred to as the "inferior" binary. For documentation on GDB see the [GDB site](#).

The following steps show you how to debug using the GNU project debugger.

- 1. Configure your build system to construct the companion debug filesystem:**

In your `local.conf` file, set the following:

```
IMAGE_GEN_DEBUGFS = "1"
IMAGE_FSTYPES_DEBUGFS = "tar.bz2"
```

These options cause the OpenEmbedded build system to generate a special companion filesystem fragment, which contains the matching source and debug symbols to your deployable filesystem. The build system does this by looking at what is in the deployed filesystem, and pulling the corresponding `-dbg` packages.

The companion debug filesystem is not a complete filesystem, but only contains the debug fragments. This filesystem must be combined with the full filesystem for debugging. Subsequent steps in this procedure show how to combine the partial filesystem with the full filesystem.

- 2. Configure the system to include `gdbserver` in the target filesystem:**

Make the following addition in either your `local.conf` file or in an image recipe:

```
IMAGE_INSTALL_append = " gdbserver"
```

The change makes sure the `gdbserver` package is included.

- 3. Build the environment:**

Use the following command to construct the image and the companion Debug Filesystem:

```
$ bitbake image
```

Build the cross GDB component and make it available for debugging. Build the SDK that matches the image. Building the SDK is best for a production build that can be used later for debugging, especially during long term maintenance:

```
$ bitbake -c populate_sdk image
```

Alternatively, you can build the minimal toolchain components that match the target. Doing so creates a smaller than typical SDK and only contains a minimal set of components with which to build simple test applications, as well as run the debugger:

```
$ bitbake meta-toolchain
```

A final method is to build Gdb itself within the build system:

```
$ bitbake gdb-cross-architecture
```

Doing so produces a temporary copy of `cross-gdb` you can use for debugging during development. While this is the quickest approach, the two previous methods in this step are better when considering long-term maintenance strategies.

Note

If you run `bitbake gdb-cross`, the OpenEmbedded build system suggests the actual image (e.g. `gdb-cross-i586`). The suggestion is usually the actual name you want to use.

- 4. Set up the debugfs**

Run the following commands to set up the debugfs:

```
$ mkdir debugfs
$ cd debugfs
$ tar xvfj build-dir/tmp-glibc/deploy/images/machine/image.rootfs.tar.bz2
$ tar xvfj build-dir/tmp-glibc/deploy/images/machine/image-dbg.rootfs.tar.bz2
```

5. Set up GDB

Install the SDK (if you built one) and then source the correct environment file. Sourcing the environment file puts the SDK in your `PATH` environment variable.

If you are using the build system, Gdb is located in `build-dir/tmp/sysroots/host/usr/bin/architecture/architecture-gdb`

6. Boot the target:

For information on how to run QEMU, see the [QEMU Documentation](#).

Note

Be sure to verify that your host can access the target via TCP.

7. Debug a program:

Debugging a program involves running gdbserver on the target and then running Gdb on the host. The example in this step debugs `gzip`:

```
root@qemux86:~# gdbserver localhost:1234 /bin/gzip -help
```

For additional gdbserver options, see the [GDB Server Documentation](#).

After running gdbserver on the target, you need to run Gdb on the host and configure it and connect to the target. Use these commands:

```
$ cd directory-holding-the-debugfs-directory
$ arch-gdb

(gdb) set sysroot debugfs
(gdb) set substitute-path /usr/src/debug debugfs/usr/src/debug
(gdb) target remote IP-of-target:1234
```

At this point, everything should automatically load (i.e. matching binaries, symbols and headers).

Note

The Gdb `set` commands in the previous example can be placed into the users `~/.gdbinit` file. Upon starting, Gdb automatically runs whatever commands are in that file.

8. Deploying without a full image rebuild:

In many cases, during development you want a quick method to deploy a new binary to the target and debug it, without waiting for a full image build.

One approach to solving this situation is to just build the component you want to debug. Once you have built the component, copy the executable directly to both the target and the host `debugfs`.

If the binary is processed through the debug splitting in OpenEmbedded, you should also copy the debug items (i.e. `.debug` contents and corresponding `/usr/src/debug` files) from the work directory. Here is an example:

```
$ bitbake bash
$ bitbake -c devshell bash
$ cd ..
$ scp packages-split/bash/bin/bash target:/bin/bash
$ cp -a packages-split/bash-dbg/* path/debugfs
```

7.30.14. Debugging with the GNU Project Debugger (GDB) on the Target¶

The previous section addressed using GDB remotely for debugging purposes, which is the most usual case due to the inherent hardware limitations on many embedded devices. However, debugging in the target hardware itself is also possible with more powerful devices. This section describes what you need to do in order to support using GDB to debug on the target hardware.

To support this kind of debugging, you need do the following:

- Ensure that GDB is on the target. You can do this by adding "gdb" to `IMAGE_INSTALL`:

```
IMAGE_INSTALL_append = " gdb"
```

Alternatively, you can add "tools-debug" to `IMAGE_FEATURES`:

```
IMAGE_FEATURES_append = " tools-debug"
```

- Ensure that debug symbols are present. You can make sure these symbols are present by installing `-dbg`:

```
IMAGE_INSTALL_append = " packagename-dbg"
```

Alternatively, you can do the following to include all the debug symbols:

```
IMAGE_FEATURES_append = " dbg-pkgs"
```

Note

To improve the debug information accuracy, you can reduce the level of optimization used by the compiler. For example, when adding the following line to your `local.conf` file, you will reduce optimization from FULL_OPTIMIZATION of "-O2" to DEBUG_OPTIMIZATION of "-O -fno-omit-frame-pointer":

```
DEBUG_BUILD = "1"
```

Consider that this will reduce the application's performance and is recommended only for debugging purposes.

7.30.15. Other Debugging Tips¶

Here are some other tips that you might find useful:

- When adding new packages, it is worth watching for undesirable items making their way into compiler command lines. For example, you do not want references to local system files like `/usr/lib/` or `/usr/include/`.
- If you want to remove the `psplash` boot splashscreen, add `psplash=false` to the kernel command line. Doing so prevents `psplash` from loading and thus allows you to see the console. It is also possible to switch out of the splashscreen by switching the virtual console (e.g. `Fn+Left` or `Fn+Right` on a Zaurus).
- Removing `TMPDIR` (usually `tmp/`, within the Build Directory) can often fix temporary build issues. Removing `TMPDIR` is usually a relatively cheap operation, because task output will be cached in SSTATE_DIR (usually `sstate-cache/`, which is also in the Build Directory).

Note

Removing `TMPDIR` might be a workaround rather than a fix. Consequently, trying to determine the underlying cause of an issue before removing the directory is a good idea.

- Understanding how a feature is used in practice within existing recipes can be very helpful. It is recommended that you configure some method that allows you to quickly search through files.

Using GNU Grep, you can use the following shell function to recursively search through common recipe-related files, skipping binary files, `.git` directories, and the Build Directory (assuming its name starts with "build"):

```
g() {
    grep -Ir \
        --exclude-dir=.git \
        --exclude-dir='build*' \
        --include='*.bb*' \
        --include='*.inc*' \
        --include='*.conf*' \
        --include='*.py*' \
        "$@"
}
```

Following are some usage examples:

```
$ g FOO # Search recursively for "FOO"
$ g -i foo # Search recursively for "foo", ignoring case
$ g -w FOO # Search recursively for "FOO" as a word, ignoring e.g. "FOOBAR"
```

If figuring out how some feature works requires a lot of searching, it might indicate that the documentation should be extended or improved. In such cases, consider filing a documentation bug using the Yocto Project implementation of Bugzilla. For information on how to submit a bug against the Yocto Project, see the Yocto Project Bugzilla wiki page and the "Submitting a Defect Against the Yocto Project" section.

Note

The manuals might not be the right place to document variables that are purely internal and have a limited scope (e.g. internal variables used to implement a single `.bbclass` file).

7.31. Making Changes to the Yocto Project¶

Because the Yocto Project is an open-source, community-based project, you can effect changes to the project. This section presents procedures that show you how to submit a defect against the project and how to submit a change.

7.31.1. Submitting a Defect Against the Yocto Project¶

Use the Yocto Project implementation of Bugzilla to submit a defect (bug) against the Yocto Project. For additional information on this implementation of Bugzilla see the "Yocto Project Bugzilla" section in the Yocto Project Reference Manual. For more detail on any of the following steps, see the Yocto Project Bugzilla wiki page.

Use the following general steps to submit a bug"

1. Open the Yocto Project implementation of [Bugzilla](#).
2. Click "File a Bug" to enter a new bug.
3. Choose the appropriate "Classification", "Product", and "Component" for which the bug was found. Bugs for the Yocto Project fall into one of several classifications, which in turn break down into several products and components. For example, for a bug against the `meta-intel` layer, you would choose "Build System, Metadata & Runtime", "BSPs", and "bsps-meta-intel", respectively.
4. Choose the "Version" of the Yocto Project for which you found the bug (e.g. 3.0.1).
5. Determine and select the "Severity" of the bug. The severity indicates how the bug impacted your work.
6. Choose the "Hardware" that the bug impacts.
7. Choose the "Architecture" that the bug impacts.
8. Choose a "Documentation change" item for the bug. Fixing a bug might or might not affect the Yocto Project documentation. If you are unsure of the impact to the documentation, select "Don't Know".
9. Provide a brief "Summary" of the bug. Try to limit your summary to just a line or two and be sure to capture the essence of the bug.
10. Provide a detailed "Description" of the bug. You should provide as much detail as you can about the context, behavior, output, and so forth that surrounds the bug. You can even attach supporting files for output from logs by using the "Add an attachment" button.
11. Click the "Submit Bug" button submit the bug. A new Bugzilla number is assigned to the bug and the defect is logged in the bug tracking system.

Once you file a bug, the bug is processed by the Yocto Project Bug Triage Team and further details concerning the bug are assigned (e.g. priority and owner). You are the "Submitter" of the bug and any further categorization, progress, or comments on the bug result in Bugzilla sending you an automated email concerning the particular change or progress to the bug.

7.31.2. Submitting a Change to the Yocto Project¶

Contributions to the Yocto Project and OpenEmbedded are very welcome. Because the system is extremely configurable and flexible, we recognize that developers will want to extend, configure or optimize it for their specific uses.

The Yocto Project uses a mailing list and a patch-based workflow that is similar to the Linux kernel but contains important differences. In general, a mailing list exists through which you can submit patches. You should send patches to the appropriate mailing list so that they can be reviewed and merged by the appropriate maintainer. The specific mailing list you need to use depends on the location of the code you are changing. Each component (e.g. layer) should have a `README` file that indicates where to send the changes and which process to follow.

You can send the patch to the mailing list using whichever approach you feel comfortable with to generate the patch. Once sent, the patch is usually reviewed by the community at large. If somebody has concerns with the patch, they will usually voice their concern over the mailing list. If a patch does not receive any negative reviews, the maintainer of the affected layer typically takes the patch, tests it, and then based on successful testing, merges the patch.

The "poky" repository, which is the Yocto Project's reference build environment, is a hybrid repository that contains several individual pieces (e.g. BitBake, Metadata, documentation, and so forth) built using the `combo-layer` tool. The upstream location used for submitting changes varies by component:

- **Core Metadata:** Send your patch to the [openembedded-core](#) mailing list. For example, a change to anything under the `meta` or `scripts` directories should be sent to this mailing list.
- **BitBake:** For changes to BitBake (i.e. anything under the `bitbake` directory), send your patch to the [bitbake-devel](#) mailing list.
- **"meta-*" trees:** These trees contain Metadata. Use the [poky](#) mailing list.

For changes to other layers hosted in the Yocto Project source repositories (i.e. `yoctoproject.org`), tools, and the Yocto Project documentation, use the [Yocto Project](#) general mailing list.

Note

Sometimes a layer's documentation specifies to use a particular mailing list. If so, use that list.

For additional recipes that do not fit into the core Metadata, you should determine which layer the recipe should go into and submit the change in the manner recommended by the documentation (e.g. the `README` file) supplied with the layer. If in doubt, please ask on the Yocto general mailing list or on the `openembedded-devel` mailing list.

You can also push a change upstream and request a maintainer to pull the change into the component's upstream repository. You do this by pushing to a contribution repository that is upstream. See the ["Git Workflows and the Yocto Project"](#) section in the Yocto Project Overview and Concepts Manual for additional concepts on working in the Yocto Project development environment.

Two commonly used testing repositories exist for OpenEmbedded-Core:

- **"ross/mut" branch:** The "mut" (master-under-test) tree exists in the `poky-contrib` repository in the [Yocto Project source repositories](#).
- **"master-next" branch:** This branch is part of the main "poky" repository in the Yocto Project source repositories.

Maintainers use these branches to test submissions prior to merging patches. Thus, you can get an idea of the status of a patch based on whether the patch has been merged into one of these branches.

Note

This system is imperfect and changes can sometimes get lost in the flow. Asking about the status of a patch or change is reasonable if the change has been idle for a while with no feedback. The Yocto Project does have plans to use [Patchwork](#) to track the status of patches and also to automatically preview patches.

The following sections provide procedures for submitting a change.

7.31.2.1. Using Scripts to Push a Change Upstream and Request a Pull

Follow this procedure to push a change to an upstream "contrib" Git repository:

Note

You can find general Git information on how to push a change upstream in the [Git Community Book](#).

1. **Make Your Changes Locally:** Make your changes in your local Git repository. You should make small, controlled, isolated changes. Keeping changes small and isolated aids review, makes merging/rebasing easier and keeps the change history clean should anyone need to refer to it in future.
2. **Stage Your Changes:** Stage your changes by using the `git add` command on each file you changed.
3. **Commit Your Changes:** Commit the change by using the `git commit` command. Make sure your commit information follows standards by following these accepted conventions:
 - Be sure to include a "Signed-off-by:" line in the same style as required by the Linux kernel. Adding this line signifies that you, the submitter, have agreed to the Developer's Certificate of Origin 1.1 as follows:


```
Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the best
    of my knowledge, is covered under an appropriate open source
    license and I have the right under that license to submit that
    work with modifications, whether created in whole or in part
    by me, under the same open source license (unless I am
    permitted to submit under a different license), as indicated
    in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```
 - Provide a single-line summary of the change. and, if more explanation is needed, provide more detail in the body of the commit. This summary is typically viewable in the "shortlist" of changes. Thus, providing something short and descriptive that gives the reader a summary of the change is useful when viewing a list of many commits. You should prefix this short description with the recipe name (if changing a recipe), or else with the short form path to the file being changed.
 - For the body of the commit message, provide detailed information that describes what you changed, why you made the change, and the approach you used. It might also be helpful if you mention how you tested the change. Provide as much detail as you can in the body of the commit message.

Note

You do not need to provide a more detailed explanation of a change if the change is minor to the point of the single line summary providing all the information.

- If the change addresses a specific bug or issue that is associated with a bug-tracking ID, include a reference to that ID in your detailed description. For example, the Yocto Project uses a specific convention for bug references - any commit that addresses a specific bug should use the following form for the detailed description. Be sure to use the actual bug-tracking ID from Bugzilla for *bug-id*:

```
Fixes [YOCTO #bug-id]

detailed description of change
```

4. **Push Your Commits to a "Contrib" Upstream:** If you have arranged for permissions to push to an upstream contrib repository, push the change to that repository:

```
$ git push upstream_remote_repo local_branch_name
```

For example, suppose you have permissions to push into the upstream `meta-intel-contrib` repository and you are working in a local branch named `your_name/README`. The following command pushes your local commits to the `meta-intel-contrib` upstream repository and puts the commit in a branch named `your_name/README`:

```
$ git push meta-intel-contrib your_name/README
```

5. **Determine Who to Notify:** Determine the maintainer or the mailing list that you need to notify for the change.

Before submitting any change, you need to be sure who the maintainer is or what mailing list that you need to notify. Use either these methods to find out:

- **Maintenance File:** Examine the `maintainers.inc` file, which is located in the [Source Directory](#) at `meta/conf/distro/include`, to see who is responsible for code.
- **Search by File:** Using [Git](#), you can enter the following command to bring up a short list of all commits against a specific file:

```
git shortlog -- filename
```

Just provide the name of the file for which you are interested. The information returned is not ordered by history but does include a list of everyone who has committed grouped by name. From the list, you can see who is responsible for the bulk of the changes against the file.

- **Examine the List of Mailing Lists:** For a list of the Yocto Project and related mailing lists, see the "[Mailing lists](#)" section in the Yocto Project Reference Manual.

6. **Make a Pull Request:** Notify the maintainer or the mailing list that you have pushed a change by making a pull request.

The Yocto Project provides two scripts that conveniently let you generate and send pull requests to the Yocto Project. These scripts are `create-pull-request` and `send-pull-request`. You can find these scripts in the `scripts` directory within the [Source Directory](#) (e.g. `~/poky/scripts`).

Using these scripts correctly formats the requests without introducing any whitespace or HTML formatting. The maintainer that receives your patches either directly or through the mailing list needs to be able to save and apply them directly from your emails. Using these scripts is the preferred method for sending patches.

First, create the pull request. For example, the following command runs the script, specifies the upstream repository in the `contrib` directory into which you pushed the change, and provides a subject line in the created patch files:

```
$ ~/poky/scripts/create-pull-request -u meta-intel-contrib -s "Updated Manual Section Reference in README"
```

Running this script forms `*.patch` files in a folder named `pull-PID` in the current directory. One of the patch files is a cover letter.

Before running the `send-pull-request` script, you must edit the cover letter patch to insert information about your change. After editing the cover letter, send the pull request. For example, the following command runs the script and specifies the patch directory and email address. In this example, the email address is a mailing list:

```
$ ~/poky/scripts/send-pull-request -p ~/meta-intel/pull-10565 -t meta-intel@yoctoproject.org
```

You need to follow the prompts as the script is interactive.

Note

For help on using these scripts, simply provide the `-h` argument as follows:

```
$ poky/scripts/create-pull-request -h
$ poky/scripts/send-pull-request -h
```

7.31.2.2. Using Email to Submit a Patch¶

You can submit patches without using the `create-pull-request` and `send-pull-request` scripts described in the previous section. However, keep in mind, the preferred method is to use the scripts.

Depending on the components changed, you need to submit the email to a specific mailing list. For some guidance on which mailing list to use, see the [list](#) at the beginning of this section. For a description of all the available mailing lists, see the "[Mailing Lists](#)" section in the Yocto Project Reference Manual.

Here is the general procedure on how to submit a patch through email without using the scripts:

1. **Make Your Changes Locally:** Make your changes in your local Git repository. You should make small, controlled, isolated changes. Keeping changes small and isolated aids review, makes merging/rebasing easier and keeps the change history clean should anyone need to refer to it in future.
2. **Stage Your Changes:** Stage your changes by using the `git add` command on each file you changed.
3. **Commit Your Changes:** Commit the change by using the `git commit --signoff` command. Using the `--signoff` option identifies you as the person making the change and also satisfies the Developer's Certificate of Origin (DCO) shown earlier.

When you form a commit, you must follow certain standards established by the Yocto Project development team. See [Step 3](#) in the previous section for information on how to provide commit information that meets Yocto Project commit message standards.

4. **Format the Commit:** Format the commit into an email message. To format commits, use the `git format-patch` command. When you provide the command, you must include a revision list or a number of patches as part of the command. For example, either of these two commands takes your most recent single commit and formats it as an email message in the current directory:

```
$ git format-patch -1
```

or

```
$ git format-patch HEAD~
```

After the command is run, the current directory contains a numbered `.patch` file for the commit.

If you provide several commits as part of the command, the `git format-patch` command produces a series of numbered files in the current directory – one for each commit. If you have more than one patch, you should also use the `--cover` option with the command, which generates a cover letter as the first "patch" in the series. You can then edit the cover letter to provide a description for the series of patches. For information on the `git format-patch` command, see `GIT_FORMAT_PATCH(1)` displayed using the `man git-format-patch` command.

Note

If you are or will be a frequent contributor to the Yocto Project or to OpenEmbedded, you might consider requesting a contrib area and the necessary associated rights.

5. Import the Files Into Your Mail Client: Import the files into your mail client by using the `git send-email` command.

Note

In order to use `git send-email`, you must have the proper Git packages installed on your host. For Ubuntu, Debian, and Fedora the package is `git-email`.

The `git send-email` command sends email by using a local or remote Mail Transport Agent (MTA) such as `msmtp`, `sendmail`, or through a direct `smtp` configuration in your Git `~/.gitconfig` file. If you are submitting patches through email only, it is very important that you submit them without any whitespace or HTML formatting that either you or your mailer introduces. The maintainer that receives your patches needs to be able to save and apply them directly from your emails. A good way to verify that what you are sending will be applicable by the maintainer is to do a dry run and send them to yourself and then save and apply them as the maintainer would.

The `git send-email` command is the preferred method for sending your patches using email since there is no risk of compromising whitespace in the body of the message, which can occur when you use your own mail client. The command also has several options that let you specify recipients and perform further editing of the email message. For information on how to use the `git send-email` command, see `GIT-SEND-EMAIL(1)` displayed using the `man git-send-email` command.

7.32. Working With Licenses¶

As mentioned in the "[Licensing](#)" section in the Yocto Project Overview and Concepts Manual, open source projects are open to the public and they consequently have different licensing structures in place. This section describes the mechanism by which the [OpenEmbedded build system](#) tracks changes to licensing text and covers how to maintain open source license compliance during your project's lifecycle. The section also describes how to enable commercially licensed recipes, which by default are disabled.

7.32.1. Tracking License Changes¶

The license of an upstream project might change in the future. In order to prevent these changes going unnoticed, the `LIC_FILES_CHKSUM` variable tracks changes to the license text. The checksums are validated at the end of the configure step, and if the checksums do not match, the build will fail.

7.32.1.1. Specifying the `LIC_FILES_CHKSUM` Variable¶

The `LIC_FILES_CHKSUM` variable contains checksums of the license text in the source code for the recipe. Following is an example of how to specify `LIC_FILES_CHKSUM`:

```
LIC_FILES_CHKSUM = "file://COPYING;md5=xxxx \
file://licfile1.txt;beginline=5;endline=29;md5=yyyy \
file://licfile2.txt;endline=50;md5=zzzz \
..."
```

Notes

- When using "beginline" and "endline", realize that line numbering begins with one and not zero. Also, the included lines are inclusive (i.e. lines five through and including 29 in the previous example for `licfile1.txt`).
- When a license check fails, the selected license text is included as part of the QA message. Using this output, you can determine the exact start and finish for the needed license text.

The build system uses the `$` variable as the default directory when searching files listed in `LIC_FILES_CHKSUM`. The previous example employs the default directory.

Consider this next example:

```
LIC_FILES_CHKSUM = "file://src/ls.c;beginline=5;endline=16;\
md5=bb14ed3c4cda583abc85401304b5cd4e"
```

```
LIC_FILES_CHKSUM = "file://${WORKDIR}/license.html;md5=5c94767cedb5d6987c902ac850ded2c6"
```

The first line locates a file in `${S}/src/ls.c` and isolates lines five through 16 as license text. The second line refers to a file in `WORKDIR`.

Note that `LIC_FILES_CHKSUM` variable is mandatory for all recipes, unless the `LICENSE` variable is set to "CLOSED".

7.32.1.2. Explanation of Syntax¶

As mentioned in the previous section, the `LIC_FILES_CHKSUM` variable lists all the important files that contain the license text for the source code. It is possible to specify a checksum for an entire file, or a specific section of a file (specified by beginning and ending line numbers with the "beginline" and "endline" parameters, respectively). The latter is useful for source files with a license notice header, README documents, and so forth. If you do not use the "beginline" parameter, then it is assumed that the text begins on the first line of the file. Similarly, if you do not use the "endline" parameter, it is assumed that the license text ends with the last line of the file.

The "md5" parameter stores the md5 checksum of the license text. If the license text changes in any way as compared to this parameter then a mismatch occurs. This mismatch triggers a build failure and notifies the developer. Notification allows the developer to review and address the license text changes. Also note that if a mismatch occurs during the build, the correct md5 checksum is placed in the build log and can be easily copied to the recipe.

There is no limit to how many files you can specify using the `LIC_FILES_CHKSUM` variable. Generally, however, every project requires a few specifications for license tracking. Many projects have a "COPYING" file that stores the license information for all the source code files. This practice allows you to just track the "COPYING" file as long as it is kept up to date.

Tips

- If you specify an empty or invalid "md5" parameter, `BitBake` returns an md5 mis-match error and displays the correct "md5" parameter value during the build. The correct parameter is also captured in the build log.
- If the whole file contains only license text, you do not need to use the "beginline" and "endline" parameters.

7.32.2. Enabling Commercially Licensed Recipes¶

By default, the OpenEmbedded build system disables components that have commercial or other special licensing requirements. Such requirements are defined on a recipe-by-recipe basis through the `LICENSE_FLAGS` variable definition in the affected recipe. For instance, the `poky/meta/recipes-multimedia/gstreamer/gst-plugins-ugly` recipe contains the following statement:

```
LICENSE_FLAGS = "commercial"
```

Here is a slightly more complicated example that contains both an explicit recipe name and version (after variable expansion):

```
LICENSE_FLAGS = "license_${PN}_${PV}"
```

In order for a component restricted by a `LICENSE_FLAGS` definition to be enabled and included in an image, it needs to have a matching entry in the global `LICENSE_FLAGS_WHITELIST` variable, which is a variable typically defined in your `local.conf` file. For example, to enable the `poky/meta/recipes-multimedia/gstreamer/gst-plugins-ugly` package, you could add either the string "commercial_gst-plugins-ugly" or the more general string "commercial" to `LICENSE_FLAGS_WHITELIST`. See the "License Flag Matching" section for a full explanation of how `LICENSE_FLAGS` matching works. Here is the example:

```
LICENSE_FLAGS_WHITELIST = "commercial_gst-plugins-ugly"
```

Likewise, to additionally enable the package built from the recipe containing `LICENSE_FLAGS = "license_${PN}_${PV}"`, and assuming that the actual recipe name was `emgd_1.10.bb`, the following string would enable that package as well as the original `gst-plugins-ugly` package:

```
LICENSE_FLAGS_WHITELIST = "commercial_gst-plugins-ugly license_emgd_1.10"
```

As a convenience, you do not need to specify the complete license string in the whitelist for every package. You can use an abbreviated form, which consists of just the first portion or portions of the license string before the initial underscore character or characters. A partial string will match any license that contains the given string as the first portion of its license. For example, the following whitelist string will also match both of the packages previously mentioned as well as any other packages that have licenses starting with "commercial" or "license".

```
LICENSE_FLAGS_WHITELIST = "commercial license"
```

7.32.2.1. License Flag Matching¶

License flag matching allows you to control what recipes the OpenEmbedded build system includes in the build. Fundamentally, the build system attempts to match `LICENSE_FLAGS` strings found in recipes against `LICENSE_FLAGS_WHITELIST` strings found in the whitelist. A match causes the build system to include a recipe in the build, while failure to find a match causes the build system to exclude a recipe.

In general, license flag matching is simple. However, understanding some concepts will help you correctly and effectively use matching.

Before a flag defined by a particular recipe is tested against the contents of the whitelist, the expanded string `_${PN}` is appended to the flag. This expansion makes each `LICENSE_FLAGS` value recipe-specific. After expansion, the string is then matched against

the whitelist. Thus, specifying `LICENSE_FLAGS = "commercial"` in recipe "foo", for example, results in the string `"commercial_foo"`. And, to create a match, that string must appear in the whitelist.

Judicious use of the `LICENSE_FLAGS` strings and the contents of the `LICENSE_FLAGS_WHITELIST` variable allows you a lot of flexibility for including or excluding recipes based on licensing. For example, you can broaden the matching capabilities by using license flags string subsets in the whitelist.

Note

When using a string subset, be sure to use the part of the expanded string that precedes the appended underscore character (e.g. `usethispart_1.3`, `usethispart_1.4`, and so forth).

For example, simply specifying the string "commercial" in the whitelist matches any expanded `LICENSE_FLAGS` definition that starts with the string "commercial" such as "commercial_foo" and "commercial_bar", which are the strings the build system automatically generates for hypothetical recipes named "foo" and "bar" assuming those recipes simply specify the following:

```
LICENSE_FLAGS = "commercial"
```

Thus, you can choose to exhaustively enumerate each license flag in the whitelist and allow only specific recipes into the image, or you can use a string subset that causes a broader range of matches to allow a range of recipes into the image.

This scheme works even if the `LICENSE_FLAGS` string already has `_${PN}` appended. For example, the build system turns the license flag "commercial_1.2_foo" into "commercial_1.2_foo_foo" and would match both the general "commercial" and the specific "commercial_1.2_foo" strings found in the whitelist, as expected.

Here are some other scenarios:

- You can specify a versioned string in the recipe such as "commercial_foo_1.2" in a "foo" recipe. The build system expands this string to "commercial_foo_1.2_foo". Combine this license flag with a whitelist that has the string "commercial" and you match the flag along with any other flag that starts with the string "commercial".
- Under the same circumstances, you can use "commercial_foo" in the whitelist and the build system not only matches "commercial_foo_1.2" but also matches any license flag with the string "commercial_foo", regardless of the version.
- You can be very specific and use both the package and version parts in the whitelist (e.g. "commercial_foo_1.2") to specifically match a versioned recipe.

7.32.2.2. Other Variables Related to Commercial Licenses¶

Other helpful variables related to commercial license handling exist and are defined in the `poky/meta/conf/distro/include/default-distrovars.inc` file:

```
COMMERCIAL_AUDIO_PLUGINS ?= ""
COMMERCIAL_VIDEO_PLUGINS ?= ""
```

If you want to enable these components, you can do so by making sure you have statements similar to the following in your `local.conf` configuration file:

```
COMMERCIAL_AUDIO_PLUGINS = "gst-plugins-ugly-mad \
gst-plugins-ugly-mpegaudioparse"
COMMERCIAL_VIDEO_PLUGINS = "gst-plugins-ugly-mpeg2dec \
gst-plugins-ugly-mpegstream gst-plugins-bad-mpegvideoparse"
LICENSE_FLAGS_WHITELIST = "commercial_gst-plugins-ugly commercial_gst-plugins-bad commercial_qmmp"
```

Of course, you could also create a matching whitelist for those components using the more general "commercial" in the whitelist, but that would also enable all the other packages with `LICENSE_FLAGS` containing "commercial", which you may or may not want:

```
LICENSE_FLAGS_WHITELIST = "commercial"
```

Specifying audio and video plugins as part of the `COMMERCIAL_AUDIO_PLUGINS` and `COMMERCIAL_VIDEO_PLUGINS` statements (along with the enabling `LICENSE_FLAGS_WHITELIST`) includes the plugins or components into built images, thus adding support for media formats or components.

7.32.3. Maintaining Open Source License Compliance During Your Product's Lifecycle¶

One of the concerns for a development organization using open source software is how to maintain compliance with various open source licensing during the lifecycle of the product. While this section does not provide legal advice or comprehensively cover all scenarios, it does present methods that you can use to assist you in meeting the compliance requirements during a software release.

With hundreds of different open source licenses that the Yocto Project tracks, it is difficult to know the requirements of each and every license. However, the requirements of the major FLOSS licenses can begin to be covered by assuming that three main areas of concern exist:

- Source code must be provided.
- License text for the software must be provided.
- Compilation scripts and modifications to the source code must be provided.

There are other requirements beyond the scope of these three and the methods described in this section (e.g. the mechanism through which source code is distributed).

As different organizations have different methods of complying with open source licensing, this section is not meant to imply that there is only one single way to meet your compliance obligations, but rather to describe one method of achieving compliance. The remainder of this section describes methods supported to meet the previously mentioned three requirements. Once you take steps to meet these requirements, and prior to releasing images, sources, and the build system, you should audit all artifacts to ensure completeness.

Note

The Yocto Project generates a license manifest during image creation that is located in `${DEPLOY_DIR}/licenses/image_name-datestamp` to assist with any audits.

7.32.3.1. Providing the Source Code¶

Compliance activities should begin before you generate the final image. The first thing you should look at is the requirement that tops the list for most compliance groups - providing the source. The Yocto Project has a few ways of meeting this requirement.

One of the easiest ways to meet this requirement is to provide the entire `DL_DIR` used by the build. This method, however, has a few issues. The most obvious is the size of the directory since it includes all sources used in the build and not just the source used in the released image. It will include toolchain source, and other artifacts, which you would not generally release. However, the more serious issue for most companies is accidental release of proprietary software. The Yocto Project provides an `archiver` class to help avoid some of these concerns.

Before you employ `DL_DIR` or the `archiver` class, you need to decide how you choose to provide source. The source `archiver` class can generate tarballs and SRPMs and can create them with various levels of compliance in mind.

One way of doing this (but certainly not the only way) is to release just the source as a tarball. You can do this by adding the following to the `local.conf` file found in the [Build Directory](#):

```
INHERIT += "archiver"
ARCHIVER_MODE[src] = "original"
```

During the creation of your image, the source from all recipes that deploy packages to the image is placed within subdirectories of `DEPLOY_DIR/sources` based on the `LICENSE` for each recipe. Releasing the entire directory enables you to comply with requirements concerning providing the unmodified source. It is important to note that the size of the directory can get large.

A way to help mitigate the size issue is to only release tarballs for licenses that require the release of source. Let us assume you are only concerned with GPL code as identified by running the following script:

```
# Script to archive a subset of packages matching specific license(s)
# Source and license files are copied into sub folders of package folder
# Must be run from build folder
#!/bin/bash
src_release_dir="source-release"
mkdir -p $src_release_dir
for a in tmp/deploy/sources/*; do
    for d in $a/*; do
        # Get package name from path
        p=`basename $d`
        p=${p%-*}
        p=${p%-*}
        # Only archive GPL packages (update *GPL* regex for your license check)
        numfiles=`ls tmp/deploy/licenses/$p/*GPL* 2> /dev/null | wc -l`
        if [ $numfiles -gt 1 ]; then
            echo Archiving $p
            mkdir -p $src_release_dir/$p/source
            cp $d/* $src_release_dir/$p/source 2> /dev/null
            mkdir -p $src_release_dir/$p/license
            cp tmp/deploy/licenses/$p/* $src_release_dir/$p/license 2> /dev/null
        fi
    done
done
```

At this point, you could create a tarball from the `gpl_source_release` directory and provide that to the end user. This method would be a step toward achieving compliance with section 3a of GPLv2 and with section 6 of GPLv3.

7.32.3.2. Providing License Text¶

One requirement that is often overlooked is inclusion of license text. This requirement also needs to be dealt with prior to generating the final image. Some licenses require the license text to accompany the binary. You can achieve this by adding the following to your `local.conf` file:

```
COPY_LIC_MANIFEST = "1"
COPY_LIC_DIRS = "1"
LICENSE_CREATE_PACKAGE = "1"
```

Adding these statements to the configuration file ensures that the licenses collected during package generation are included on your image.

Note

Setting all three variables to "1" results in the image having two copies of the same license file. One copy resides in `/usr/share/common-licenses` and the other resides in `/usr/share/license`.

The reason for this behavior is because `COPY_LIC_DIRS` and `COPY_LIC_MANIFEST` add a copy of the license when the image is built but do not offer a path for adding licenses for newly installed packages to an image. `LICENSE_CREATE_PACKAGE` adds a separate package and an upgrade path for adding licenses to an image.

As the source `archiver` class has already archived the original unmodified source that contains the license files, you would have already met the requirements for inclusion of the license information with source as defined by the GPL and other open source licenses.

7.32.3.3. Providing Compilation Scripts and Source Code Modifications¶

At this point, we have addressed all we need to prior to generating the image. The next two requirements are addressed during the final packaging of the release.

By releasing the version of the OpenEmbedded build system and the layers used during the build, you will be providing both compilation scripts and the source code modifications in one step.

If the deployment team has a [BSP layer](#) and a distro layer, and those those layers are used to patch, compile, package, or modify (in any way) any open source software included in your released images, you might be required to release those layers under section 3 of GPLv2 or section 1 of GPLv3. One way of doing that is with a clean checkout of the version of the Yocto Project and layers used during your build. Here is an example:

```
# We built using the zeus branch of the poky repo
$ git clone -b zeus git://git.yoctoproject.org/poky
$ cd poky
# We built using the release_branch for our layers
$ git clone -b release_branch git://git.mycompany.com/meta-my-bsp-layer
$ git clone -b release_branch git://git.mycompany.com/meta-my-software-layer
# clean up the .git repos
$ find . -name ".git" -type d -exec rm -rf {} \;
```

One thing a development organization might want to consider for end-user convenience is to modify `meta-poky/conf/bblayers.conf.sample` to ensure that when the end user utilizes the released build system to build an image, the development organization's layers are included in the `bblayers.conf` file automatically:

```
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

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

BBLAYERS ?= " \
    ##OEROOT##/meta \
    ##OEROOT##/meta-poky \
    ##OEROOT##/meta-yocto-bsp \
    ##OEROOT##/meta-my-layer \
"
```

Creating and providing an archive of the [Metadata](#) layers (recipes, configuration files, and so forth) enables you to meet your requirements to include the scripts to control compilation as well as any modifications to the original source.

7.32.4. Copying Licenses that Do Not Exist¶

Some packages, such as the linux-firmware package, have many licenses that are not in any way common. You can avoid adding a lot of these types of common license files, which are only applicable to a specific package, by using the [NO_GENERIC_LICENSE](#) variable. Using this variable also avoids QA errors when you use a non-common, non-CLOSED license in a recipe.

The following is an example that uses the `LICENSE.Abilis.txt` file as the license from the fetched source:

```
NO_GENERIC_LICENSE[Firmware-Abilis] = "LICENSE.Abilis.txt"
```

7.33. Using the Error Reporting Tool¶

The error reporting tool allows you to submit errors encountered during builds to a central database. Outside of the build environment, you can use a web interface to browse errors, view statistics, and query for errors. The tool works using a client-server system where the client portion is integrated with the installed Yocto Project [Source Directory](#) (e.g. `poky`). The server receives the information collected and saves it in a database.

A live instance of the error reporting server exists at <http://errors.yoctoproject.org>. This server exists so that when you want to get help with build failures, you can submit all of the information on the failure easily and then point to the URL in your bug report or send an email to the mailing list.

Note

If you send error reports to this server, the reports become publicly visible.

7.33.1. Enabling and Using the Tool¶

By default, the error reporting tool is disabled. You can enable it by inheriting the `report-error` class by adding the following statement to the end of your `local.conf` file in your [Build Directory](#).

```
INHERIT += "report-error"
```

By default, the error reporting feature stores information in `${LOG_DIR}/error-report`. However, you can specify a directory to use by adding the following to your `local.conf` file:

```
ERR_REPORT_DIR = "path"
```

Enabling error reporting causes the build process to collect the errors and store them in a file as previously described. When the build system encounters an error, it includes a command as part of the console output. You can run the command to send the error file to the server. For example, the following command sends the errors to an upstream server:

```
$ send-error-report /home/brandusa/project/poky/build/tmp/log/error-report/error_report_201403141617.txt
```

In the previous example, the errors are sent to a public database available at <http://errors.yoctoproject.org>, which is used by the entire community. If you specify a particular server, you can send the errors to a different database. Use the following command for more information on available options:

```
$ send-error-report --help
```

When sending the error file, you are prompted to review the data being sent as well as to provide a name and optional email address. Once you satisfy these prompts, the command returns a link from the server that corresponds to your entry in the database. For example, here is a typical link:

```
http://errors.yoctoproject.org/Errors/Details/9522/
```

Following the link takes you to a web interface where you can browse, query the errors, and view statistics.

7.33.2. Disabling the Tool

To disable the error reporting feature, simply remove or comment out the following statement from the end of your `local.conf` file in your [Build Directory](#).

```
INHERIT += "report-error"
```

7.33.3. Setting Up Your Own Error Reporting Server

If you want to set up your own error reporting server, you can obtain the code from the Git repository at <http://git.yoctoproject.org/cgit.cgi/error-report-web/>. Instructions on how to set it up are in the README document.

7.34. Using Wayland and Weston

[Wayland](#) is a computer display server protocol that provides a method for compositing window managers to communicate directly with applications and video hardware and expects them to communicate with input hardware using other libraries. Using Wayland with supporting targets can result in better control over graphics frame rendering than an application might otherwise achieve.

The Yocto Project provides the Wayland protocol libraries and the reference [Weston](#) compositor as part of its release. You can find the integrated packages in the meta layer of the [Source Directory](#). Specifically, you can find the recipes that build both Wayland and Weston at `meta/recipes-graphics/wayland`.

You can build both the Wayland and Weston packages for use only with targets that accept the [Mesa 3D and Direct Rendering Infrastructure](#), which is also known as Mesa DRI. This implies that you cannot build and use the packages if your target uses, for example, the Intel® Embedded Media and Graphics Driver (Intel® EMGD) that overrides Mesa DRI.

Note

Due to lack of EGL support, Weston 1.0.3 will not run directly on the emulated QEMU hardware. However, this version of Weston will run under X emulation without issues.

This section describes what you need to do to implement Wayland and use the Weston compositor when building an image for a supporting target.

7.34.1. Enabling Wayland in an Image

To enable Wayland, you need to enable it to be built and enable it to be included (installed) in the image.

7.34.1.1. Building

To cause Mesa to build the `wayland-egl` platform and Weston to build Wayland with Kernel Mode Setting ([KMS](#)) support, include the "wayland" flag in the `DISTRO_FEATURES` statement in your `local.conf` file:

```
DISTRO_FEATURES_append = " wayland"
```

Note

If X11 has been enabled elsewhere, Weston will build Wayland with X11 support

7.34.1.2. Installing

To install the Wayland feature into an image, you must include the following `CORE_IMAGE_EXTRA_INSTALL` statement in your `local.conf` file:

```
CORE_IMAGE_EXTRA_INSTALL += "wayland weston"
```

7.34.2. Running Weston

To run Weston inside X11, enabling it as described earlier and building a Sato image is sufficient. If you are running your image under Sato, a Weston Launcher appears in the "Utility" category.

Alternatively, you can run Weston through the command-line interpreter (CLI), which is better suited for development work. To run Weston under the CLI, you need to do the following after your image is built:

1. Run these commands to export `XDG_RUNTIME_DIR`:

```
mkdir -p /tmp/$USER-weston
chmod 0700 /tmp/$USER-weston
export XDG_RUNTIME_DIR=/tmp/$USER-weston
```

2. Launch Weston in the shell:

```
weston
```

Chapter 8. Using the Quick EMULATOR (QEMU)¶

- [8.1. Overview](#)
- [8.2. Running QEMU](#)
- [8.3. Switching Between Consoles](#)
- [8.4. Removing the Splash Screen](#)
- [8.5. Disabling the Cursor Grab](#)
- [8.6. Running Under a Network File System \(NFS\) Server](#)
- [8.7. QEMU CPU Compatibility Under KVM](#)
- [8.8. QEMU Performance](#)
- [8.9. QEMU Command-Line Syntax](#)
- [8.10. `runqemu` Command-Line Options](#)

The Yocto Project uses an implementation of the Quick EMULATOR (QEMU) Open Source project as part of the Yocto Project development "tool set". This chapter provides both procedures that show you how to use the Quick EMULATOR (QEMU) and other QEMU information helpful for development purposes.

8.1. Overview¶

Within the context of the Yocto Project, QEMU is an emulator and virtualization machine that allows you to run a complete image you have built using the Yocto Project as just another task on your build system. QEMU is useful for running and testing images and applications on supported Yocto Project architectures without having actual hardware. Among other things, the Yocto Project uses QEMU to run automated Quality Assurance (QA) tests on final images shipped with each release.

Note

This implementation is not the same as QEMU in general.

This section provides a brief reference for the Yocto Project implementation of QEMU.

For official information and documentation on QEMU in general, see the following references:

- **[QEMU Website](#)**: The official website for the QEMU Open Source project.
- **[Documentation](#)**: The QEMU user manual.

8.2. Running QEMU¶

To use QEMU, you need to have QEMU installed and initialized as well as have the proper artifacts (i.e. image files and root filesystems) available. Follow these general steps to run QEMU:

1. **Install QEMU**: QEMU is made available with the Yocto Project a number of ways. One method is to install a Software Development Kit (SDK). See "[The QEMU Emulator](#)" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual for information on how to install QEMU.
2. **Setting Up the Environment**: How you set up the QEMU environment depends on how you installed QEMU:
 - If you cloned the `poky` repository or you downloaded and unpacked a Yocto Project release tarball, you can source the build environment script (i.e. [oe-init-build-env](#)):


```
$ cd ~/poky
$ source oe-init-build-env
```
 - If you installed a cross-toolchain, you can run the script that initializes the toolchain. For example, the following commands run the initialization script from the default `poky_sdk` directory:


```
~ /poky_sdk/environment-setup-core2-64-poky-linux
```
3. **Ensure the Artifacts are in Place**: You need to be sure you have a pre-built kernel that will boot in QEMU. You also need the target root filesystem for your target machine's architecture:
 - If you have previously built an image for QEMU (e.g. `qemuarm64`, `qemuarm`, and so forth), then the artifacts are in place in your [Build Directory](#).
 - If you have not built an image, you can go to the [machines/qemu](#) area and download a pre-built image that matches your architecture and can be run on QEMU.

See the "Extracting the Root Filesystem" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual for information on how to extract a root filesystem.

4. **Run QEMU:** The basic `runqemu` command syntax is as follows:

```
$ runqemu [option ] [...]
```

Based on what you provide on the command line, `runqemu` does a good job of figuring out what you are trying to do. For example, by default, QEMU looks for the most recently built image according to the timestamp when it needs to look for an image. Minimally, through the use of options, you must provide either a machine name, a virtual machine image (`*wic.vmdk`), or a kernel image (`*.bin`).

Here are some additional examples to help illustrate further QEMU:

- This example starts QEMU with `MACHINE` set to "qemux86". Assuming a standard [Build Directory](#), `runqemu` automatically finds the `bzImage-qemux86.bin` image file and the `core-image-minimal-qemux86-20140707074611.rootfs.ext3` (assuming the current build created a `core-image-minimal` image).

Note

When more than one image with the same name exists, QEMU finds and uses the most recently built image according to the timestamp.

```
$ runqemu qemux86
```

- This example produces the exact same results as the previous example. This command, however, specifically provides the image and root filesystem type.

```
$ runqemu qemux86 core-image-minimal ext3
```

- This example specifies to boot an initial RAM disk image and to enable audio in QEMU. For this case, `runqemu` set the internal variable `FSTYPE` to "cpio.gz". Also, for audio to be enabled, an appropriate driver must be installed (see the previous description for the `audio` option for more information).

```
$ runqemu qemux86 ramfs audio
```

- This example does not provide enough information for QEMU to launch. While the command does provide a root filesystem type, it must also minimally provide a `MACHINE`, `KERNEL`, or `VM` option.

```
$ runqemu ext3
```

- This example specifies to boot a virtual machine image (`.wic.vmdk` file). From the `.wic.vmdk`, `runqemu` determines the QEMU architecture (`MACHINE`) to be "qemux86" and the root filesystem type to be "vmdk".

```
$ runqemu /home/scott-lenovo/vm/core-image-minimal-qemux86.wic.vmdk
```

8.3. Switching Between Consoles¶

When booting or running QEMU, you can switch between supported consoles by using `Ctrl+Alt+number`. For example, `Ctrl+Alt+3` switches you to the serial console as long as that console is enabled. Being able to switch consoles is helpful, for example, if the main QEMU console breaks for some reason.

Note

Usually, "2" gets you to the main console and "3" gets you to the serial console.

8.4. Removing the Splash Screen¶

You can remove the splash screen when QEMU is booting by using `Alt+left`. Removing the splash screen allows you to see what is happening in the background.

8.5. Disabling the Cursor Grab¶

The default QEMU integration captures the cursor within the main window. It does this since standard mouse devices only provide relative input and not absolute coordinates. You then have to break out of the grab using the "Ctrl+Alt" key combination. However, the Yocto Project's integration of QEMU enables the wacom USB touch pad driver by default to allow input of absolute coordinates. This default means that the mouse can enter and leave the main window without the grab taking effect leading to a better user experience.

8.6. Running Under a Network File System (NFS) Server¶

One method for running QEMU is to run it on an NFS server. This is useful when you need to access the same file system from both the build and the emulated system at the same time. It is also worth noting that the system does not need root privileges to run. It uses a user space NFS server to avoid that. Follow these steps to set up for running QEMU using an NFS server.

1. **Extract a Root Filesystem:** Once you are able to run QEMU in your environment, you can use the `runqemu-extract-sdk` script, which is located in the `scripts` directory along with the `runqemu` script.

The `runqemu-extract-sdk` takes a root filesystem tarball and extracts it into a location that you specify. Here is an example that takes a file system and extracts it to a directory named `test-nfs`:

```
runqemu-extract-sdk ./tmp/deploy/images/qemux86/core-image-sato-qemux86.tar.bz2 test-nfs
```

2. **Start QEMU:** Once you have extracted the file system, you can run `runqemu` normally with the additional location of the file system. You can then also make changes to the files within `./test-nfs` and see those changes appear in the image in real time. Here is an example using the `qemux86` image:

```
runqemu qemux86 ./test-nfs
```

Note

Should you need to start, stop, or restart the NFS share, you can use the following commands:

- The following command starts the NFS share:

```
runqemu-export-rootfs start file-system-Location
```

- The following command stops the NFS share:

```
runqemu-export-rootfs stop file-system-Location
```

- The following command restarts the NFS share:

```
runqemu-export-rootfs restart file-system-Location
```

8.7. QEMU CPU Compatibility Under KVM¶

By default, the QEMU build compiles for and targets 64-bit and x86 Intel® Core™2 Duo processors and 32-bit x86 Intel® Pentium® II processors. QEMU builds for and targets these CPU types because they display a broad range of CPU feature compatibility with many commonly used CPUs.

Despite this broad range of compatibility, the CPUs could support a feature that your host CPU does not support. Although this situation is not a problem when QEMU uses software emulation of the feature, it can be a problem when QEMU is running with KVM enabled. Specifically, software compiled with a certain CPU feature crashes when run on a CPU under KVM that does not support that feature. To work around this problem, you can override QEMU's runtime CPU setting by changing the `QB_CPU_KVM` variable in `qemuboot.conf` in the [Build Directory's](#) `deploy/image` directory. This setting specifies a `-cpu` option passed into QEMU in the `runqemu` script. Running `qemu -cpu help` returns a list of available supported CPU types.

8.8. QEMU Performance¶

Using QEMU to emulate your hardware can result in speed issues depending on the target and host architecture mix. For example, using the `qemux86` image in the emulator on an Intel-based 32-bit (x86) host machine is fast because the target and host architectures match. On the other hand, using the `qemuarm` image on the same Intel-based host can be slower. But, you still achieve faithful emulation of ARM-specific issues.

To speed things up, the QEMU images support using `distcc` to call a cross-compiler outside the emulated system. If you used `runqemu` to start QEMU, and the `distccd` application is present on the host system, any BitBake cross-compiling toolchain available from the build system is automatically used from within QEMU simply by calling `distcc`. You can accomplish this by defining the cross-compiler variable (e.g. `export CC="distcc"`). Alternatively, if you are using a suitable SDK image or the appropriate stand-alone toolchain is present, the toolchain is also automatically used.

Note

Several mechanisms exist that let you connect to the system running on the QEMU emulator:

- QEMU provides a framebuffer interface that makes standard consoles available.
- Generally, headless embedded devices have a serial port. If so, you can configure the operating system of the running image to use that port to run a console. The connection uses standard IP networking.
- SSH servers exist in some QEMU images. The `core-image-sato` QEMU image has a Dropbear secure shell (SSH) server that runs with the root password disabled. The `core-image-full-cmdline` and `core-image-lsb` QEMU images have OpenSSH instead of Dropbear. Including these SSH servers allow you to use standard `ssh` and `scp` commands. The `core-image-minimal` QEMU image, however, contains no SSH server.
- You can use a provided, user-space NFS server to boot the QEMU session using a local copy of the root filesystem on the host. In order to make this connection, you must extract a root filesystem tarball by using the `runqemu-extract-sdk` command. After running the command, you must then point the `runqemu` script to the extracted directory instead of a root filesystem image file. See the ["Running Under a Network File System \(NFS\) Server"](#) section for more information.

8.9. QEMU Command-Line Syntax¶

The basic `runqemu` command syntax is as follows:

```
$ runqemu [option ] [...]
```

Based on what you provide on the command line, `runqemu` does a good job of figuring out what you are trying to do. For example, by default, QEMU looks for the most recently built image according to the timestamp when it needs to look for an image. Minimally, through the use of options, you must provide either a machine name, a virtual machine image (`*wic.vmdk`), or a kernel image (`*.bin`).

Following is the command-line help output for the `runqemu` command:

```
$ runqemu --help

Usage: you can run this script with any valid combination
of the following environment variables (in any order):
  KERNEL - the kernel image file to use
  ROOTFS - the rootfs image file or nfsroot directory to use
  MACHINE - the machine name (optional, autodetected from KERNEL filename if unspecified)
Simplified QEMU command-line options can be passed with:
  nographic - disable video console
  serial - enable a serial console on /dev/ttyS0
  slirp - enable user networking, no root privileges is required
  kvm - enable KVM when running x86/x86_64 (VT-capable CPU required)
  kvm-vhost - enable KVM with vhost when running x86/x86_64 (VT-capable CPU required)
  publicvnc - enable a VNC server open to all hosts
  audio - enable audio
  [*/]ovmf* - OVMF firmware file or base name for booting with UEFI
  tcpserial=<port> - specify tcp serial port number
  biosdir=<dir> - specify custom bios dir
  biosfilename=<filename> - specify bios filename
  qemuparams=<xyz> - specify custom parameters to QEMU
  bootparams=<xyz> - specify custom kernel parameters during boot
  help, -h, --help: print this text

Examples:
runqemu
runqemu qemuarm
runqemu tmp/deploy/images/qemuarm
runqemu tmp/deploy/images/qemux86/<qemuboot.conf>
runqemu qemux86-64 core-image-sato ext4
runqemu qemux86-64 wic-image-minimal wic
runqemu path/to/bzImage-qemux86.bin path/to/nfsrootdir/ serial
runqemu qemux86 iso/hddimg/wic.vmdk/wic.qcow2/wic.vdi/ramfs/cpio.gz...
runqemu qemux86 qemuparams="-m 256"
runqemu qemux86 bootparams="psplash=false"
runqemu path/to/<image>-<machine>.wic
runqemu path/to/<image>-<machine>.wic.vmdk
```

8.10. runqemu Command-Line Options¶

Following is a description of `runqemu` options you can provide on the command line:

Tip

If you do provide some "illegal" option combination or perhaps you do not provide enough in the way of options, `runqemu` provides appropriate error messaging to help you correct the problem.

- **QEMUARCH**: The QEMU machine architecture, which must be "qemuarm", "qemuarm64", "qemumips", "qemumips64", "qemuppc", "qemux86", or "qemux86-64".
- **VM**: The virtual machine image, which must be a `.wic.vmdk` file. Use this option when you want to boot a `.wic.vmdk` image. The image filename you provide must contain one of the following strings: "qemux86-64", "qemux86", "qemuarm", "qemumips64", "qemumips", "qemuppc", or "qemush4".
- **ROOTFS**: A root filesystem that has one of the following filetype extensions: "ext2", "ext3", "ext4", "jffs2", "nfs", or "btrfs". If the filename you provide for this option uses "nfs", it must provide an explicit root filesystem path.
- **KERNEL**: A kernel image, which is a `.bin` file. When you provide a `.bin` file, `runqemu` detects it and assumes the file is a kernel image.
- **MACHINE**: The architecture of the QEMU machine, which must be one of the following: "qemux86", "qemux86-64", "qemuarm", "qemuarm64", "qemumips", "qemumips64", or "qemuppc". The **MACHINE** and **QEMUARCH** options are basically identical. If you do not provide a **MACHINE** option, `runqemu` tries to determine it based on other options.
- **ramfs**: Indicates you are booting an initial RAM disk (initramfs) image, which means the **FSTYPE** is `cpio.gz`.
- **iso**: Indicates you are booting an ISO image, which means the **FSTYPE** is `.iso`.
- **nographic**: Disables the video console, which sets the console to "ttyS0". This option is useful when you have logged into a server and you do not want to disable forwarding from the X Window System (X11) to your workstation or laptop.
- **serial**: Enables a serial console on `/dev/ttyS0`.
- **biosdir**: Establishes a custom directory for BIOS, VGA BIOS and keymaps.
- **biosfilename**: Establishes a custom BIOS name.

- `qemuparams="\xyz\"`: Specifies custom QEMU parameters. Use this option to pass options other than the simple "kvm" and "serial" options.
- `bootparams="\xyz\"`: Specifies custom boot parameters for the kernel.
- `audio`: Enables audio in QEMU. The *MACHINE* option must be either "qemux86" or "qemux86-64" in order for audio to be enabled. Additionally, the `snd_intel8x0` or `snd_ens1370` driver must be installed in linux guest.
- `slirp`: Enables "slirp" networking, which is a different way of networking that does not need root access but also is not as easy to use or comprehensive as the default.
- `kvm`: Enables KVM when running "qemux86" or "qemux86-64" QEMU architectures. For KVM to work, all the following conditions must be met:
 - Your *MACHINE* must be either `qemux86` or `qemux86-64`.
 - Your build host has to have the KVM modules installed, which are `/dev/kvm`.
 - The build host `/dev/kvm` directory has to be both writable and readable.
- `kvm-vhost`: Enables KVM with VHOST support when running "qemux86" or "qemux86-64" QEMU architectures. For KVM with VHOST to work, the following conditions must be met:
 - `kvm` option conditions must be met.
 - Your build host has to have virtio net device, which are `/dev/vhost-net`.
 - The build host `/dev/vhost-net` directory has to be either readable or writable and "slirp-enabled".
- `publicvnc`: Enables a VNC server open to all hosts.

Yocto Project Application Development and the Extensible Software Development Kit (eSDK)



Chapter 9. Introduction¶

9.1. Introduction

9.1.1. The Cross-Development Toolchain

9.1.2. Sysroots

9.1.3. The QEMU Emulator

9.2. SDK Development Model

9.1. Introduction¶

Welcome to the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual. This manual provides information that explains how to use both the Yocto Project extensible and standard SDKs to develop applications and images.

Note

Prior to the 2.0 Release of the Yocto Project, application development was primarily accomplished through the use of the Application Development Toolkit (ADT) and the availability of stand-alone cross-development toolchains and other tools. With the 2.1 Release of the Yocto Project, application development has transitioned to within a tool-rich extensible SDK and the more traditional standard SDK.

All SDKs consist of the following:

- **Cross-Development Toolchain**: This toolchain contains a compiler, debugger, and various miscellaneous tools.
- **Libraries, Headers, and Symbols**: The libraries, headers, and symbols are specific to the image (i.e. they match the image).
- **Environment Setup Script**: This `*.sh` file, once run, sets up the cross-development environment by defining variables and preparing for SDK use.

Additionally, an extensible SDK has tools that allow you to easily add new applications and libraries to an image, modify the source of an existing component, test changes on the target hardware, and easily integrate an application into the [OpenEmbedded build system](#).

You can use an SDK to independently develop and test code that is destined to run on some target machine. SDKs are completely self-contained. The binaries are linked against their own copy of `libc`, which results in no dependencies on the target system. To achieve this, the pointer to the dynamic loader is configured at install time since that path cannot be dynamically altered. This is the reason for a wrapper around the `populate_sdk` and `populate_sdk_ext` archives.

Another feature for the SDKs is that only one set of cross-compiler toolchain binaries are produced for any given architecture. This feature takes advantage of the fact that the target hardware can be passed to `gcc` as a set of compiler options. Those options are set up by the environment script and contained in variables such as `CC` and `LD`. This reduces the space needed for the tools. Understand, however, that every target still needs a sysroot because those binaries are target-specific.

The SDK development environment consists of the following:

- The self-contained SDK, which is an architecture-specific cross-toolchain and matching sysroots (target and native) all built by the OpenEmbedded build system (e.g. the SDK). The toolchain and sysroots are based on a [Metadata](#) configuration and extensions, which allows you to cross-develop on the host machine for the target hardware. Additionally, the extensible SDK contains the `devtool` functionality.
- The Quick EMULATOR (QEMU), which lets you simulate target hardware. QEMU is not literally part of the SDK. You must build and include this emulator separately. However, QEMU plays an important role in the development process that revolves around use of the SDK.

In summary, the extensible and standard SDK share many features. However, the extensible SDK has powerful development tools to help you more quickly develop applications. Following is a table that summarizes the primary differences between the standard and extensible SDK types when considering which to build:

Feature	Standard SDK	Extensible SDK
Toolchain	Yes	Yes*
Debugger	Yes	Yes*
Size	100+ MBytes	1+ GBytes (or 300+ MBytes for minimal w/toolchain)
devtool	No	Yes
Build Images	No	Yes
Updateable	No	Yes
Managed Sysroot**	No	Yes
Installed Packages	No***	Yes****
Construction	Packages	Shared State

* Extensible SDK contains the toolchain and debugger if `SDK_EXT_TYPE` is "full" or `SDK_INCLUDE_TOOLCHAIN` is "yes".

** Sysroot is managed through the use of `devtool`. Thus, it is less likely that you will corrupt your SDK sysroot when using `devtool`.

*** You can add runtime package management to the standard SDK but it is not supported by default.

**** You must build and make the shared state available to extensible SDK users for "packages" you want to enable users to install.

9.1.1. The Cross-Development Toolchain¶

The [Cross-Development Toolchain](#) consists of a cross-compiler, cross-linker, and cross-debugger that are used to develop user-space applications for targeted hardware. Additionally, for an extensible SDK, the toolchain also has built-in `devtool` functionality. This toolchain is created by running a SDK installer script or through a [Build Directory](#), that is based on your metadata configuration or extension for your targeted device. The cross-toolchain works with a matching target sysroot.

9.1.2. Sysroots¶

The native and target sysroots contain needed headers and libraries for generating binaries that run on the target architecture. The target sysroot is based on the target root filesystem image that is built by the OpenEmbedded build system and uses the same metadata configuration used to build the cross-toolchain.

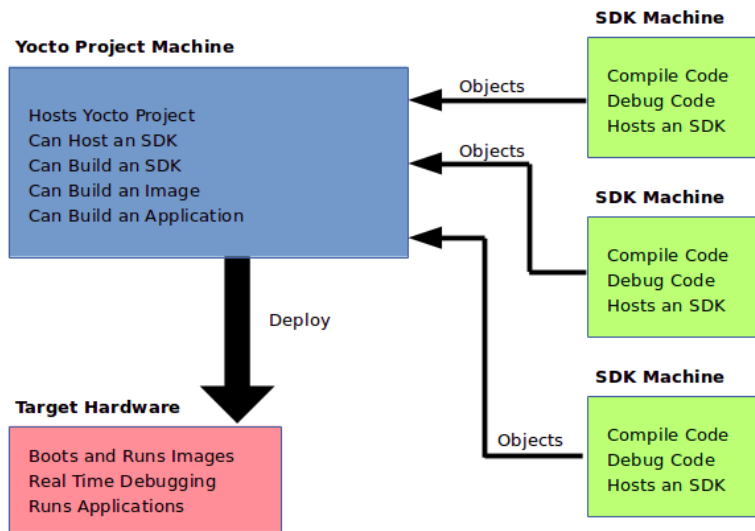
9.1.3. The QEMU Emulator¶

The QEMU emulator allows you to simulate your hardware while running your application or image. QEMU is not part of the SDK but is made available a number of different ways:

- If you have cloned the `pokey` Git repository to create a [Source Directory](#) and you have sourced the environment setup script, QEMU is installed and automatically available.
- If you have downloaded a Yocto Project release and unpacked it to create a Source Directory and you have sourced the environment setup script, QEMU is installed and automatically available.
- If you have installed the cross-toolchain tarball and you have sourced the toolchain's setup environment script, QEMU is also installed and automatically available.

9.2. SDK Development Model¶

Fundamentally, the SDK fits into the development process as follows:



The SDK is installed on any machine and can be used to develop applications, images, and kernels. An SDK can even be used by a QA Engineer or Release Engineer. The fundamental concept is that the machine that has the SDK installed does not have to be associated with the machine that has the Yocto Project installed. A developer can independently compile and test an object on their machine and then, when the object is ready for integration into an image, they can simply make it available to the machine that has the Yocto Project. Once the object is available, the image can be rebuilt using the Yocto Project to produce the modified image.

You just need to follow these general steps:

1. **Install the SDK for your target hardware:** For information on how to install the SDK, see the "[Installing the SDK](#)" section.
2. **Download or Build the Target Image:** The Yocto Project supports several target architectures and has many pre-built kernel images and root filesystem images.

If you are going to develop your application on hardware, go to the [machines](#) download area and choose a target machine area from which to download the kernel image and root filesystem. This download area could have several files in it that support development using actual hardware. For example, the area might contain `.hddimg` files that combine the kernel image with the filesystem, boot loaders, and so forth. Be sure to get the files you need for your particular development process.

If you are going to develop your application and then run and test it using the QEMU emulator, go to the [machines/qemu](#) download area. From this area, go down into the directory for your target architecture (e.g. `qemux86_64` for an Intel®-based 64-bit architecture). Download the kernel, root filesystem, and any other files you need for your process.

Note

To use the root filesystem in QEMU, you need to extract it. See the "[Extracting the Root Filesystem](#)" section for information on how to extract the root filesystem.

3. **Develop and Test your Application:** At this point, you have the tools to develop your application. If you need to separately install and use the QEMU emulator, you can go to [QEMU Home Page](#) to download and learn about the emulator. See the "[Using the Quick EMUlator \(QEMU\)](#)" chapter in the Yocto Project Development Tasks Manual for information on using QEMU within the Yocto Project.

The remainder of this manual describes how to use the extensible and standard SDKs. Information also exists in appendix form that describes how you can build, install, and modify an SDK.

Chapter 10. Using the Extensible SDK¶

[10.1. Why use the Extensible SDK and What is in It?](#)

[10.2. Installing the Extensible SDK](#)

[10.3. Running the Extensible SDK Environment Setup Script](#)

[10.4. Using devtool in Your SDK Workflow](#)

[10.4.1. Use devtool add to Add an Application](#)

[10.4.2. Use devtool modify to Modify the Source of an Existing Component](#)

[10.4.3. Use devtool upgrade to Create a Version of the Recipe that Supports a Newer Version of the Software](#)

[10.5. A Closer Look at devtool add](#)

[10.5.1. Name and Version](#)

[10.5.2. Dependency Detection and Mapping](#)

[10.5.3. License Detection](#)

[10.5.4. Adding Makefile-Only Software](#)

[10.5.5. Adding Native Tools](#)

[10.5.6. Adding Node.js Modules](#)

[10.6. Working With Recipes](#)

[10.6.1. Finding Logs and Work Files](#)

[10.6.2. Setting Configure Arguments](#)

[10.6.3. Sharing Files Between Recipes](#)

[10.6.4. Packaging](#)

[10.7. Restoring the Target Device to its Original State](#)

[10.8. Installing Additional Items Into the Extensible SDK](#)
[10.9. Applying Updates to an Installed Extensible SDK](#)
[10.10. Creating a Derivative SDK With Additional Components](#)

This chapter describes the extensible SDK and how to install it. Information covers the pieces of the SDK, how to install it, and presents a look at using the `devtool` functionality. The extensible SDK makes it easy to add new applications and libraries to an image, modify the source for an existing component, test changes on the target hardware, and ease integration into the rest of the [OpenEmbedded build system](#).

Note

For a side-by-side comparison of main features supported for an extensible SDK as compared to a standard SDK, see the "[Introduction](#)" section.

In addition to the functionality available through `devtool`, you can alternatively make use of the toolchain directly, for example from Makefile and Autotools. See the "[Using the SDK Toolchain Directly](#)," chapter for more information.

10.1. Why use the Extensible SDK and What is in It?

The extensible SDK provides a cross-development toolchain and libraries tailored to the contents of a specific image. You would use the Extensible SDK if you want a toolchain experience supplemented with the powerful set of `devtool` commands tailored for the Yocto Project environment.

The installed extensible SDK consists of several files and directories. Basically, it contains an SDK environment setup script, some configuration files, an internal build system, and the `devtool` functionality.

10.2. Installing the Extensible SDK

The first thing you need to do is install the SDK on your [Build Host](#) by running the `*.sh` installation script.

You can download a tarball installer, which includes the pre-built toolchain, the `runqemu` script, the internal build system, `devtool`, and support files from the appropriate [toolchain](#) directory within the Index of Releases. Toolchains are available for several 32-bit and 64-bit architectures with the `x86_64` directories, respectively. The toolchains the Yocto Project provides are based off the `core-image-sato` and `core-image-minimal` images and contain libraries appropriate for developing against that image.

The names of the tarball installer scripts are such that a string representing the host system appears first in the filename and then is immediately followed by a string representing the target architecture. An extensible SDK has the string "-ext" as part of the name. Following is the general form:

```
poky-glibc-host_system-image_type-arch-toolchain-ext-release_version.sh
```

Where:

`host_system` is a string representing your development system:

i686 or x86_64.

`image_type` is the image for which the SDK was built:

core-image-sato or core-image-minimal

`arch` is a string representing the tuned target architecture:

aarch64, armv5e, core2-64, i586, mips32r2, mips64, ppc7400, or cortexa8hf-neon

`release_version` is a string representing the release number of the Yocto Project:

3.0.1, 3.0.1+snapshot

For example, the following SDK installer is for a 64-bit development host system and a i586-tuned target architecture based off the SDK for `core-image-sato` and using the current 3.0.1 snapshot:

```
poky-glibc-x86_64-core-image-sato-i586-toolchain-ext-3.0.1.sh
```

Note

As an alternative to downloading an SDK, you can build the SDK installer. For information on building the installer, see the "[Building an SDK Installer](#)" section.

The SDK and toolchains are self-contained and by default are installed into the `poky_sdk` folder in your home directory. You can choose to install the extensible SDK in any location when you run the installer. However, because files need to be written under that directory during the normal course of operation, the location you choose for installation must be writable for whichever users need to use the SDK.

The following command shows how to run the installer given a toolchain tarball for a 64-bit x86 development host system and a 64-bit x86 target architecture. The example assumes the SDK installer is located in `~/Downloads/` and has execution rights.

Note

If you do not have write permissions for the directory into which you are installing the SDK, the installer notifies you and exits. For that case, set up the proper permissions in the directory and run the installer again.

```
$ ./Downloads/poky-glibc-x86_64-core-image-minimal-core2-64-toolchain-ext-2.5.sh
Poky (Yocto Project Reference Distro) Extensible SDK installer version 2.5
=====
Enter target directory for SDK (default: ~/poky_sdk):
You are about to install the SDK to "/home/scottrif/poky_sdk". Proceed [Y/n]? Y
Extracting SDK.....done
Setting it up...
Extracting buildtools...
Preparing build system...
Parsing recipes: 100% |#####| Time: 0:00:52
Initialising tasks: 100% |#####| Time: 0:00:00
Checking sstate mirror object availability: 100% |#####| Time: 0:00:00
Loading cache: 100% |#####| Time: 0:00:00
Initialising tasks: 100% |#####| Time: 0:00:00
done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.
$ . /home/scottrif/poky_sdk/environment-setup-core2-64-poky-linux
```

10.3. Running the Extensible SDK Environment Setup Script¶

Once you have the SDK installed, you must run the SDK environment setup script before you can actually use the SDK. This setup script resides in the directory you chose when you installed the SDK, which is either the default `poky_sdk` directory or the directory you chose during installation.

Before running the script, be sure it is the one that matches the architecture for which you are developing. Environment setup scripts begin with the string "environment-setup" and include as part of their name the tuned target architecture. As an example, the following commands set the working directory to where the SDK was installed and then source the environment setup script. In this example, the setup script is for an IA-based target machine using i586 tuning:

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

Running the setup script defines many environment variables needed in order to use the SDK (e.g. `PATH`, `CC`, `LD`, and so forth). If you want to see all the environment variables the script exports, examine the installation file itself.

10.4. Using devtool in Your SDK Workflow¶

The cornerstone of the extensible SDK is a command-line tool called `devtool`. This tool provides a number of features that help you build, test and package software within the extensible SDK, and optionally integrate it into an image built by the OpenEmbedded build system.

Tip

The use of `devtool` is not limited to the extensible SDK. You can use `devtool` to help you easily develop any project whose build output must be part of an image built using the build system.

The `devtool` command line is organized similarly to `Git` in that it has a number of sub-commands for each function. You can run `devtool --help` to see all the commands.

Note

See the "[devtool Quick Reference](#)" in the Yocto Project Reference Manual for a `devtool` quick reference.

Three `devtool` subcommands exist that provide entry-points into development:

- **`devtool add`**: Assists in adding new software to be built.
- **`devtool modify`**: Sets up an environment to enable you to modify the source of an existing component.
- **`devtool upgrade`**: Updates an existing recipe so that you can build it for an updated set of source files.

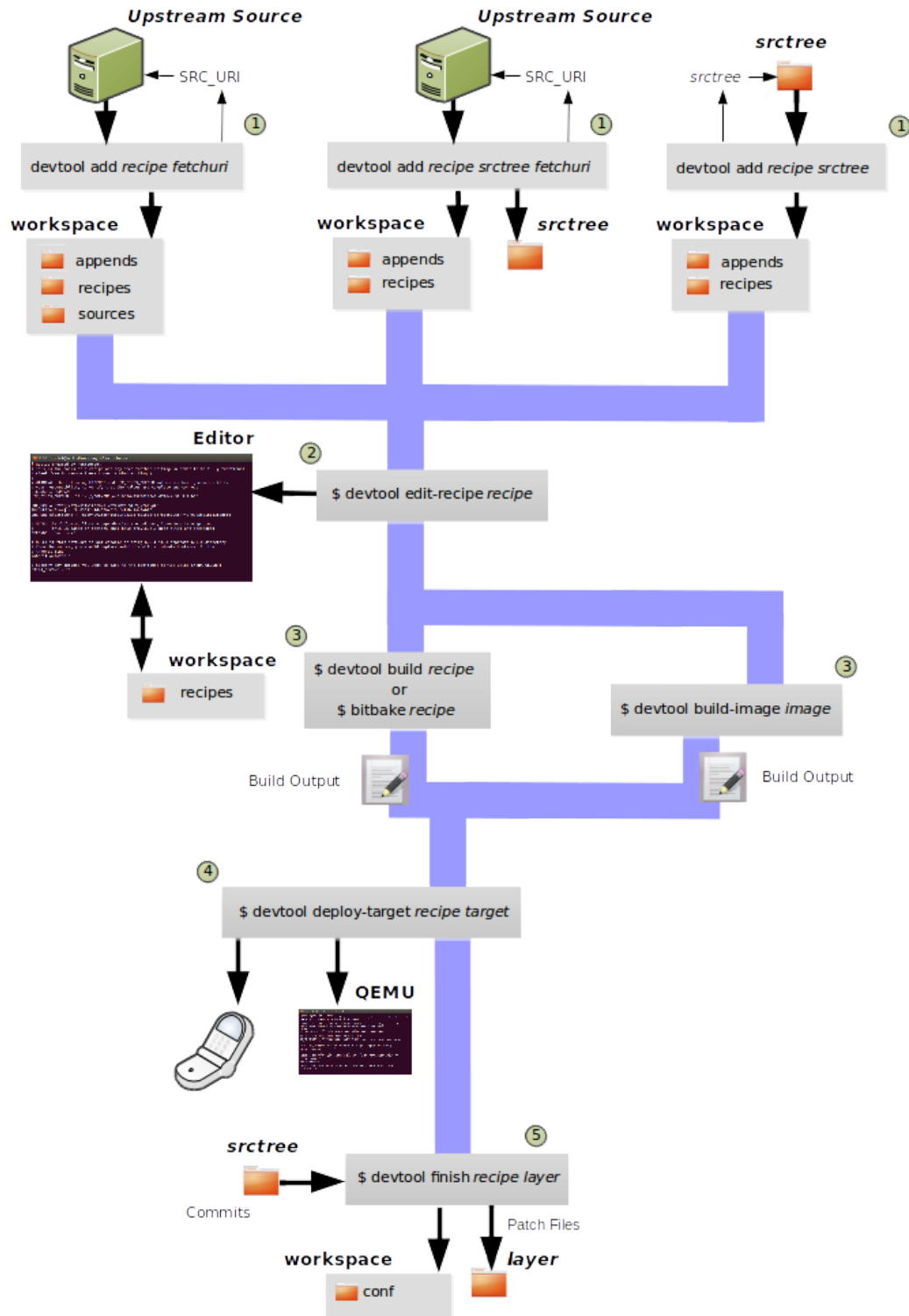
As with the build system, "recipes" represent software packages within `devtool`. When you use `devtool add`, a recipe is automatically created. When you use `devtool modify`, the specified existing recipe is used in order to determine where to get the source code and how to patch it. In both cases, an environment is set up so that when you build the recipe a source tree that is under your control is used in order to allow you to make changes to the source as desired. By default, new recipes and the source go into a "workspace" directory under the SDK.

The remainder of this section presents the `devtool add`, `devtool modify`, and `devtool upgrade` workflows.

10.4.1. Use devtool add to Add an Application¶

The `devtool add` command generates a new recipe based on existing source code. This command takes advantage of the `workspace` layer that many `devtool` commands use. The command is flexible enough to allow you to extract source code into both the workspace or a separate local Git repository and to use existing code that does not need to be extracted.

Depending on your particular scenario, the arguments and options you use with `devtool add` form different combinations. The following diagram shows common development flows you would use with the `devtool add` command:



1. **Generating the New Recipe:** The top part of the flow shows three scenarios by which you could use `devtool add` to generate a recipe based on existing source code.

In a shared development environment, it is typical for other developers to be responsible for various areas of source code. As a developer, you are probably interested in using that source code as part of your development within the Yocto Project. All you need is access to the code, a recipe, and a controlled area in which to do your work.

Within the diagram, three possible scenarios feed into the `devtool add` workflow:

- **Left:** The left scenario in the figure represents a common situation where the source code does not exist locally and needs to be extracted. In this situation, the source code is extracted to the default workspace - you do not want the files in some specific location outside of the workspace. Thus, everything you need will be located in the workspace:

```
$ devtool add recipe fetchuri
```

With this command, `devtool` extracts the upstream source files into a local Git repository within the `SOURCES` folder. The command then creates a recipe named `recipe` and a corresponding append file in the workspace. If you do not provide `recipe`, the command makes an attempt to determine the recipe name.

- **Middle:** The middle scenario in the figure also represents a situation where the source code does not exist locally. In this case, the code is again upstream and needs to be extracted to some local area - this time outside of the default workspace.

Note

If required, `devtool` always creates a Git repository locally during the extraction.

Furthermore, the first positional argument `src tree` in this case identifies where the `devtool add` command will locate the extracted code outside of the workspace. You need to specify an empty directory:

```
$ devtool add recipe src tree fetchuri
```

In summary, the source code is pulled from `fetchuri` and extracted into the location defined by `src tree` as a local Git repository.

Within workspace, `devtool` creates a recipe named `recipe` along with an associated append file.

- **Right:** The right scenario in the figure represents a situation where the `src tree` has been previously prepared outside of the `devtool` workspace.

The following command provides a new recipe name and identifies the existing source tree location:

```
$ devtool add recipe src tree
```

The command examines the source code and creates a recipe named `recipe` for the code and places the recipe into the workspace.

Because the extracted source code already exists, `devtool` does not try to relocate the source code into the workspace - only the new recipe is placed in the workspace.

Aside from a recipe folder, the command also creates an associated append folder and places an initial `*.bbappend` file within.

2. **Edit the Recipe:** You can use `devtool edit-recipe` to open up the editor as defined by the `$EDITOR` environment variable and modify the file:

```
$ devtool edit-recipe recipe
```

From within the editor, you can make modifications to the recipe that take affect when you build it later.

3. **Build the Recipe or Rebuild the Image:** The next step you take depends on what you are going to do with the new code.

If you need to eventually move the build output to the target hardware, use the following `devtool` command:

```
$ devtool build recipe
```

On the other hand, if you want an image to contain the recipe's packages from the workspace for immediate deployment onto a device (e.g. for testing purposes), you can use the `devtool build-image` command:

```
$ devtool build-image image
```

4. **Deploy the Build Output:** When you use the `devtool build` command to build out your recipe, you probably want to see if the resulting build output works as expected on the target hardware.

Note

This step assumes you have a previously built image that is already either running in QEMU or is running on actual hardware. Also, it is assumed that for deployment of the image to the target, SSH is installed in the image and, if the image is running on real hardware, you have network access to and from your development machine.

You can deploy your build output to that target hardware by using the `devtool deploy-target` command:

```
$ devtool deploy-target recipe target
```

The `target` is a live target machine running as an SSH server.

You can, of course, also deploy the image you build to actual hardware by using the `devtool build-image` command. However, `devtool` does not provide a specific command that allows you to deploy the image to actual hardware.

5. **Finish Your Work With the Recipe:** The `devtool finish` command creates any patches corresponding to commits in the local Git repository, moves the new recipe to a more permanent layer, and then resets the recipe so that the recipe is built normally rather than from the workspace.

```
$ devtool finish recipe layer
```

Note

Any changes you want to turn into patches must be committed to the Git repository in the source tree.

As mentioned, the `devtool finish` command moves the final recipe to its permanent layer.

As a final process of the `devtool finish` command, the state of the standard layers and the upstream source is restored so that you can build the recipe from those areas rather than the workspace.

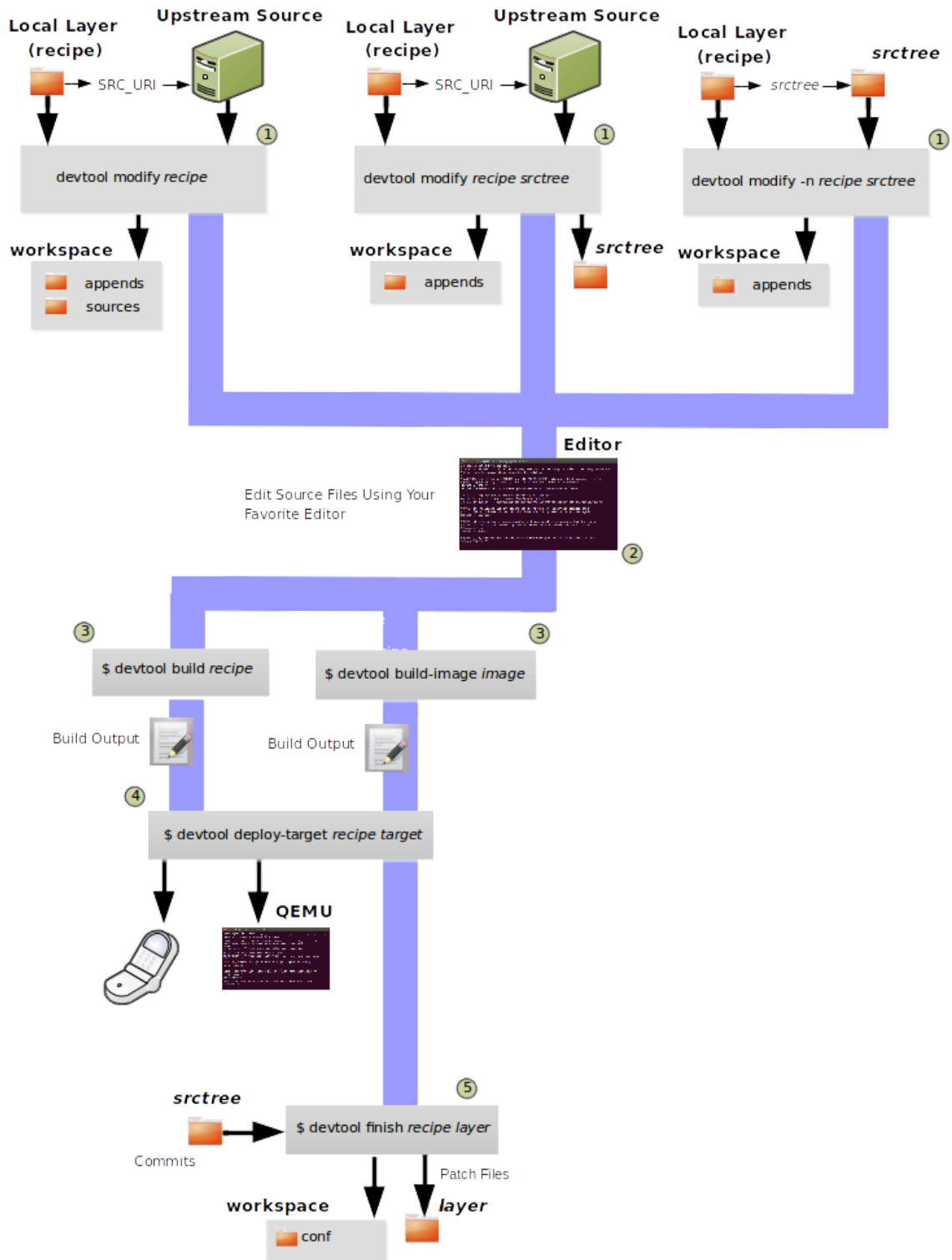
Note

You can use the `devtool reset` command to put things back should you decide you do not want to proceed with your work. If you do use this command, realize that the source tree is preserved.

10.4.2. Use `devtool modify` to Modify the Source of an Existing Component¶

The `devtool modify` command prepares the way to work on existing code that already has a local recipe in place that is used to build the software. The command is flexible enough to allow you to extract code from an upstream source, specify the existing recipe, and keep track of and gather any patch files from other developers that are associated with the code.

Depending on your particular scenario, the arguments and options you use with `devtool modify` form different combinations. The following diagram shows common development flows for the `devtool modify` command:



1. **Preparing to Modify the Code:** The top part of the flow shows three scenarios by which you could use `devtool modify` to prepare to work on source files. Each scenario assumes the following:

- The recipe exists locally in a layer external to the `devtool` workspace.
- The source files exist either upstream in an un-extracted state or locally in a previously extracted state.

The typical situation is where another developer has created a layer for use with the Yocto Project and their recipe already resides in that layer. Furthermore, their source code is readily available either upstream or locally.

- **Left:** The left scenario in the figure represents a common situation where the source code does not exist locally and it needs to be extracted from an upstream source. In this situation, the source is extracted into the default `devtool` workspace location. The recipe, in this scenario, is in its own layer outside the workspace (i.e. `meta-Layername`).

The following command identifies the recipe and, by default, extracts the source files:

```
$ devtool modify recipe
```

Once `devtool` locates the recipe, `devtool` uses the recipe's `SRC_URI` statements to locate the source code and any local patch files from other developers.

With this scenario, no `srctree` argument exists. Consequently, the default behavior of the `devtool modify` command is to extract the source files pointed to by the `SRC_URI` statements into a local Git structure. Furthermore, the location for the extracted source is the default area within the `devtool` workspace. The result is that the command sets up both the source code and an append file within the workspace while the recipe remains in its original location.

Additionally, if you have any non-patch local files (i.e. files referred to with `file://` entries in `SRC_URI` statement excluding `*.patch/` or `*.diff`), these files are copied to an `oe-local-files` folder under the newly created source tree. Copying the files here gives you a convenient area from which you can modify the files. Any changes or additions you make to those files are incorporated into the build the next time you build the software just as are other changes you might have made to the source.

- **Middle:** The middle scenario in the figure represents a situation where the source code also does not exist locally. In this case, the code is again upstream and needs to be extracted to some local area as a Git repository. The recipe, in this scenario, is again local and in its own layer outside the workspace.

The following command tells `devtool` the recipe with which to work and, in this case, identifies a local area for the extracted source files that exists outside of the default `devtool` workspace:

```
$ devtool modify recipe srctree
```

Note

You cannot provide a URL for `srctree` using the `devtool` command.

As with all extractions, the command uses the recipe's `SRC_URI` statements to locate the source files and any associated patch files. Non-patch files are copied to an `oe-local-files` folder under the newly created source tree.

Once the files are located, the command by default extracts them into `srctree`.

Within workspace, `devtool` creates an append file for the recipe. The recipe remains in its original location but the source files are extracted to the location you provide with `srctree`.

- **Right:** The right scenario in the figure represents a situation where the source tree (`srctree`) already exists locally as a previously extracted Git structure outside of the `devtool` workspace. In this example, the recipe also exists elsewhere locally in its own layer.

The following command tells `devtool` the recipe with which to work, uses the `-n` option to indicate source does not need to be extracted, and uses `srctree` to point to the previously extracted source files:

```
$ devtool modify -n recipe srctree
```

If an `oe-local-files` subdirectory happens to exist and it contains non-patch files, the files are used. However, if the subdirectory does not exist and you run the `devtool finish` command, any non-patch files that might exist next to the recipe are removed because it appears to `devtool` that you have deleted those files.

Once the `devtool modify` command finishes, it creates only an append file for the recipe in the `devtool` workspace. The recipe and the source code remain in their original locations.

2. **Edit the Source:** Once you have used the `devtool modify` command, you are free to make changes to the source files. You can use any editor you like to make and save your source code modifications.
3. **Build the Recipe or Rebuild the Image:** The next step you take depends on what you are going to do with the new code.

If you need to eventually move the build output to the target hardware, use the following `devtool` command:

```
$ devtool build recipe
```

On the other hand, if you want an image to contain the recipe's packages from the workspace for immediate deployment onto a device (e.g. for testing purposes), you can use the `devtool build-image` command:

```
$ devtool build-image image
```

4. **Deploy the Build Output:** When you use the `devtool build` command to build out your recipe, you probably want to see if the resulting build output works as expected on target hardware.

Note

This step assumes you have a previously built image that is already either running in QEMU or running on actual hardware. Also, it is assumed that for deployment of the image to the target, SSH is installed in the image and if the image is running on real hardware that you have network access to and from your development machine.

You can deploy your build output to that target hardware by using the `devtool deploy-target` command:

```
$ devtool deploy-target recipe target
```

The `target` is a live target machine running as an SSH server.

You can, of course, use other methods to deploy the image you built using the `devtool build-image` command to actual hardware. `devtool` does not provide a specific command to deploy the image to actual hardware.

5. **Finish Your Work With the Recipe:** The `devtool finish` command creates any patches corresponding to commits in the local Git repository, updates the recipe to point to them (or creates a `.bbappend` file to do so, depending on the specified destination layer), and then resets the recipe so that the recipe is built normally rather than from the workspace.

```
$ devtool finish recipe Layer
```

Note

Any changes you want to turn into patches must be staged and committed within the local Git repository before you use the `devtool finish` command.

Because there is no need to move the recipe, `devtool finish` either updates the original recipe in the original layer or the command creates a `.bbappend` file in a different layer as provided by *Layer*. Any work you did in the `oe-local-files` directory is preserved in the original files next to the recipe during the `devtool finish` command.

As a final process of the `devtool finish` command, the state of the standard layers and the upstream source is restored so that you can build the recipe from those areas rather than from the workspace.

Note

You can use the `devtool reset` command to put things back should you decide you do not want to proceed with your work. If you do use this command, realize that the source tree is preserved.

10.4.3. Use `devtool upgrade` to Create a Version of the Recipe that Supports a Newer Version of the Software¶

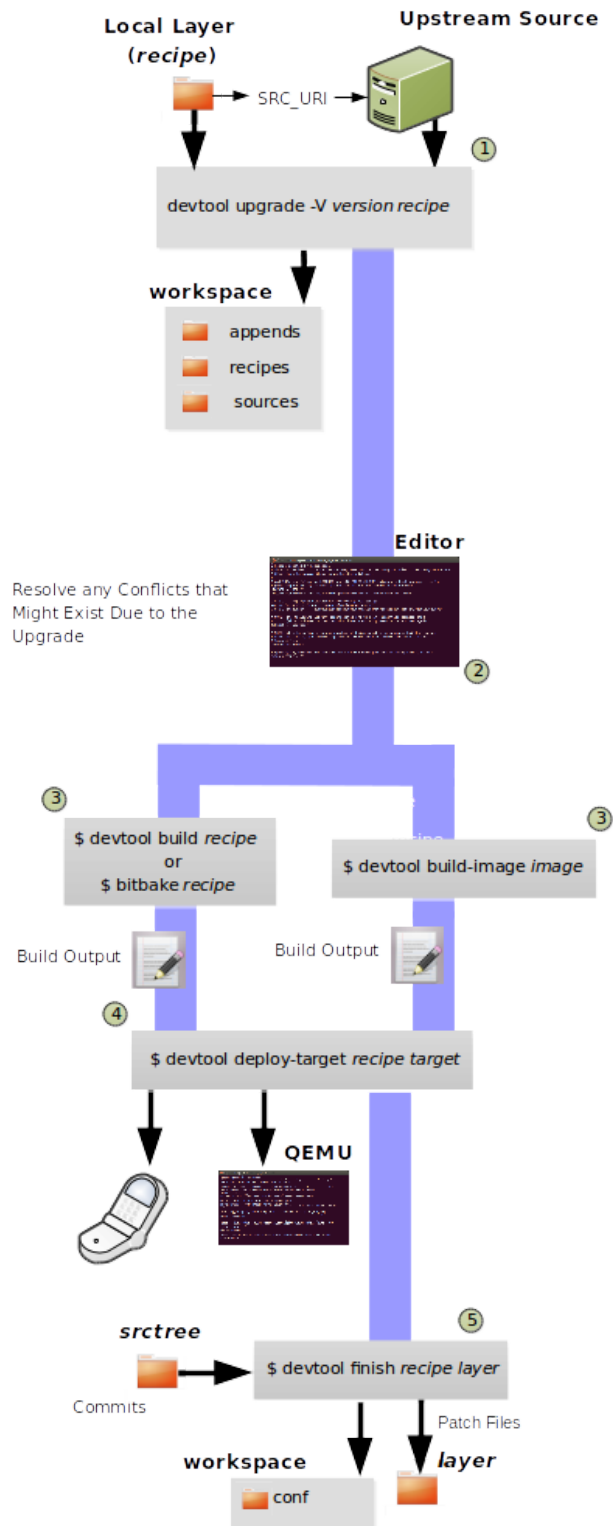
The `devtool upgrade` command upgrades an existing recipe to that of a more up-to-date version found upstream. Throughout the life of software, recipes continually undergo version upgrades by their upstream publishers. You can use the `devtool upgrade` workflow to make sure your recipes you are using for builds are up-to-date with their upstream counterparts.

Note

Several methods exist by which you can upgrade recipes - `devtool upgrade` happens to be one. You can read about all the methods by which you can upgrade recipes in the "[Upgrading Recipes](#)" section of the Yocto Project Development Tasks Manual.

The `devtool upgrade` command is flexible enough to allow you to specify source code revision and versioning schemes, extract code into or out of the `devtool workspace`, and work with any source file forms that the [fetchers](#) support.

The following diagram shows the common development flow used with the `devtool upgrade` command:



1. **Initiate the Upgrade:** The top part of the flow shows the typical scenario by which you use the `devtool upgrade` command. The following conditions exist:

- The recipe exists in a local layer external to the `devtool` workspace.
- The source files for the new release exist in the same location pointed to by `SRC_URI` in the recipe (e.g. a tarball with the new version number in the name, or as a different revision in the upstream Git repository).

A common situation is where third-party software has undergone a revision so that it has been upgraded. The recipe you have access to is likely in your own layer. Thus, you need to upgrade the recipe to use the newer version of the software:

```
$ devtool upgrade -V version recipe
```

By default, the `devtool upgrade` command extracts source code into the `SOURCES` directory in the `workspace`. If you want the code extracted to any other location, you need to provide the `srctree` positional argument with the command as follows:

```
$ devtool upgrade -V version recipe srctree
```

Note

In this example, the "-V" option specifies the new version. If you don't use "-V", the command upgrades the recipe to the latest version.

If the source files pointed to by the `SRC_URI` statement in the recipe are in a Git repository, you must provide the "-S" option and specify a revision for the software.

Once `devtool` locates the recipe, it uses the `SRC_URI` variable to locate the source code and any local patch files from other developers. The result is that the command sets up the source code, the new version of the recipe, and an append file all within the workspace.

Additionally, if you have any non-patch local files (i.e. files referred to with `file://` entries in `SRC_URI` statement excluding `*.patch/` or `*.diff`), these files are copied to an `oe-local-files` folder under the newly created source tree. Copying the files here gives you a convenient area from which you can modify the files. Any changes or additions you make to those files are incorporated into the build the next time you build the software just as are other changes you might have made to the source.

2. **Resolve any Conflicts created by the Upgrade:** Conflicts could exist due to the software being upgraded to a new version. Conflicts occur if your recipe specifies some patch files in `SRC_URI` that conflict with changes made in the new version of the software. For such cases, you need to resolve the conflicts by editing the source and following the normal `git rebase` conflict resolution process.

Before moving onto the next step, be sure to resolve any such conflicts created through use of a newer or different version of the software.

3. **Build the Recipe or Rebuild the Image:** The next step you take depends on what you are going to do with the new code.

If you need to eventually move the build output to the target hardware, use the following `devtool` command:

```
$ devtool build recipe
```

On the other hand, if you want an image to contain the recipe's packages from the workspace for immediate deployment onto a device (e.g. for testing purposes), you can use the `devtool build-image` command:

```
$ devtool build-image image
```

4. **Deploy the Build Output:** When you use the `devtool build` command or `bitbake` to build your recipe, you probably want to see if the resulting build output works as expected on target hardware.

Note

This step assumes you have a previously built image that is already either running in QEMU or running on actual hardware. Also, it is assumed that for deployment of the image to the target, SSH is installed in the image and if the image is running on real hardware that you have network access to and from your development machine.

You can deploy your build output to that target hardware by using the `devtool deploy-target` command:

```
$ devtool deploy-target recipe target
```

The *target* is a live target machine running as an SSH server.

You can, of course, also deploy the image you build using the `devtool build-image` command to actual hardware. However, `devtool` does not provide a specific command that allows you to do this.

5. **Finish Your Work With the Recipe:** The `devtool finish` command creates any patches corresponding to commits in the local Git repository, moves the new recipe to a more permanent layer, and then resets the recipe so that the recipe is built normally rather than from the workspace.

Any work you did in the `oe-local-files` directory is preserved in the original files next to the recipe during the `devtool finish` command.

If you specify a destination layer that is the same as the original source, then the old version of the recipe and associated files are removed prior to adding the new version.

```
$ devtool finish recipe layer
```

Note

Any changes you want to turn into patches must be committed to the Git repository in the source tree.

As a final process of the `devtool finish` command, the state of the standard layers and the upstream source is restored so that you can build the recipe from those areas rather than the workspace.

Note

You can use the `devtool reset` command to put things back should you decide you do not want to proceed with your work. If you do use this command, realize that the source tree is preserved.

10.5. A Closer Look at `devtool add`

The `devtool add` command automatically creates a recipe based on the source tree you provide with the command. Currently, the command has support for the following:

- Autotools (`autoconf` and `automake`)
- CMake
- Scons
- `qmake`
- Plain Makefile
- Out-of-tree kernel module
- Binary package (i.e. `--b` option)
- Node.js module
- Python modules that use `setuptools` or `distutils`

Apart from binary packages, the determination of how a source tree should be treated is automatic based on the files present within that source tree. For example, if a `CMakeLists.txt` file is found, then the source tree is assumed to be using CMake and is treated accordingly.

Note

In most cases, you need to edit the automatically generated recipe in order to make it build properly. Typically, you would go through several edit and build cycles until the recipe successfully builds. Once the recipe builds, you could use possible further iterations to test the recipe on the target device.

The remainder of this section covers specifics regarding how parts of the recipe are generated.

10.5.1. Name and Version

If you do not specify a name and version on the command line, `devtool add` uses various metadata within the source tree in an attempt to determine the name and version of the software being built. Based on what the tool determines, `devtool` sets the name of the created recipe file accordingly.

If `devtool` cannot determine the name and version, the command prints an error. For such cases, you must re-run the command and provide the name and version, just the name, or just the version as part of the command line.

Sometimes the name or version determined from the source tree might be incorrect. For such a case, you must reset the recipe:

```
$ devtool reset -n recipename
```

After running the `devtool reset` command, you need to run `devtool add` again and provide the name or the version.

10.5.2. Dependency Detection and Mapping

The `devtool add` command attempts to detect build-time dependencies and map them to other recipes in the system. During this mapping, the command fills in the names of those recipes as part of the `DEPENDS` variable within the recipe. If a dependency cannot be mapped, `devtool` places a comment in the recipe indicating such. The inability to map a dependency can result from naming not being recognized or because the dependency simply is not available. For cases where the dependency is not available, you must use the `devtool add` command to add an additional recipe that satisfies the dependency. Once you add that recipe, you need to update the `DEPENDS` variable in the original recipe to include the new recipe.

If you need to add runtime dependencies, you can do so by adding the following to your recipe:

```
RDEPENDS_${PN} += "dependency1 dependency2 ..."
```

Note

The `devtool add` command often cannot distinguish between mandatory and optional dependencies. Consequently, some of the detected dependencies might in fact be optional. When in doubt, consult the documentation or the configure script for the software the recipe is building for further details. In some cases, you might find you can substitute the dependency with an option that disables the associated functionality passed to the configure script.

10.5.3. License Detection

The `devtool add` command attempts to determine if the software you are adding is able to be distributed under a common, open-source license. If so, the command sets the `LICENSE` value accordingly. You should double-check the value added by the command against the documentation or source files for the software you are building and, if necessary, update that `LICENSE` value.

The `devtool add` command also sets the `LIC_FILES_CHKSUM` value to point to all files that appear to be license-related. Realize that license statements often appear in comments at the top of source files or within the documentation. In such cases, the command does not recognize those license statements. Consequently, you might need to amend the `LIC_FILES_CHKSUM` variable to point to one or more of those comments if present. Setting `LIC_FILES_CHKSUM` is particularly important for third-party software. The mechanism attempts to ensure correct licensing should you upgrade the recipe to a newer upstream version in future. Any change in licensing is detected and you receive an error prompting you to check the license text again.

If the `devtool add` command cannot determine licensing information, `devtool` sets the `LICENSE` value to "CLOSED" and leaves the `LIC_FILES_CHKSUM` value unset. This behavior allows you to continue with development even though the settings are unlikely to be correct in all cases. You should check the documentation or source files for the software you are building to determine the actual license.

10.5.4. Adding Makefile-Only Software¶

The use of Make by itself is very common in both proprietary and open-source software. Unfortunately, Makefiles are often not written with cross-compilation in mind. Thus, `devtool add` often cannot do very much to ensure that these Makefiles build correctly. It is very common, for example, to explicitly call `gcc` instead of using the `CC` variable. Usually, in a cross-compilation environment, `gcc` is the compiler for the build host and the cross-compiler is named something similar to `arm-poky-linux-gnueabi-gcc` and might require arguments (e.g. to point to the associated sysroot for the target machine).

When writing a recipe for Makefile-only software, keep the following in mind:

- You probably need to patch the Makefile to use variables instead of hardcoding tools within the toolchain such as `gcc` and `g++`.
- The environment in which Make runs is set up with various standard variables for compilation (e.g. `CC`, `CXX`, and so forth) in a similar manner to the environment set up by the SDK's environment setup script. One easy way to see these variables is to run the `devtool build` command on the recipe and then look in `oe-logs/run.do_compile`. Towards the top of this file, a list of environment variables exists that are being set. You can take advantage of these variables within the Makefile.
- If the Makefile sets a default for a variable using "=", that default overrides the value set in the environment, which is usually not desirable. For this case, you can either patch the Makefile so it sets the default using the "?:=" operator, or you can alternatively force the value on the `make` command line. To force the value on the command line, add the variable setting to `EXTRA_OEMAKE` or `PACKAGECONFIG_CONFARGS` within the recipe. Here is an example using `EXTRA_OEMAKE`:

```
EXTRA_OEMAKE += "'CC=${CC}' 'CXX=${CXX}'"
```

In the above example, single quotes are used around the variable settings as the values are likely to contain spaces because required default options are passed to the compiler.

- Hardcoding paths inside Makefiles is often problematic in a cross-compilation environment. This is particularly true because those hardcoded paths often point to locations on the build host and thus will either be read-only or will introduce contamination into the cross-compilation because they are specific to the build host rather than the target. Patching the Makefile to use prefix variables or other path variables is usually the way to handle this situation.
- Sometimes a Makefile runs target-specific commands such as `ldconfig`. For such cases, you might be able to apply patches that remove these commands from the Makefile.

10.5.5. Adding Native Tools¶

Often, you need to build additional tools that run on the build host as opposed to the target. You should indicate this requirement by using one of the following methods when you run `devtool add`:

- Specify the name of the recipe such that it ends with "-native". Specifying the name like this produces a recipe that only builds for the build host.
- Specify the "--also-native" option with the `devtool add` command. Specifying this option creates a recipe file that still builds for the target but also creates a variant with a "-native" suffix that builds for the build host.

Note

If you need to add a tool that is shipped as part of a source tree that builds code for the target, you can typically accomplish this by building the native and target parts separately rather than within the same compilation process. Realize though that with the "--also-native" option, you can add the tool using just one recipe file.

10.5.6. Adding Node.js Modules¶

You can use the `devtool add` command two different ways to add Node.js modules: 1) Through `npm` and, 2) from a repository or local source.

Use the following form to add Node.js modules through `npm`:

```
$ devtool add "npm://registry.npmjs.org;name=forever;version=0.15.1"
```

The name and version parameters are mandatory. Lockdown and shrinkwrap files are generated and pointed to by the recipe in order to freeze the version that is fetched for the dependencies according to the first time. This also saves checksums that are verified on future fetches. Together, these behaviors ensure the reproducibility and integrity of the build.

Notes

- You must use quotes around the URL. The `devtool add` does not require the quotes, but the shell considers ";" as a splitter between multiple commands. Thus, without the quotes, `devtool add` does not receive the other parts, which results in several "command not found" errors.
- In order to support adding Node.js modules, a `nodejs` recipe must be part of your SDK.

As mentioned earlier, you can also add Node.js modules directly from a repository or local source tree. To add modules this way, use `devtool add` in the following form:


```
$ devtool add https://github.com/diversario/node-ssdp
```

In this example, `devtool` fetches the specified Git repository, detects the code as Node.js code, fetches dependencies using `npm`, and sets `SRC_URI` accordingly.

10.6. Working With Recipes¶

When building a recipe using the `devtool build` command, the typical build progresses as follows:

1. Fetch the source
2. Unpack the source
3. Configure the source
4. Compile the source
5. Install the build output
6. Package the installed output

For recipes in the workspace, fetching and unpacking is disabled as the source tree has already been prepared and is persistent. Each of these build steps is defined as a function (task), usually with a "do_" prefix (e.g. `do_fetch`, `do_unpack`, and so forth). These functions are typically shell scripts but can instead be written in Python.

If you look at the contents of a recipe, you will see that the recipe does not include complete instructions for building the software. Instead, common functionality is encapsulated in classes inherited with the `inherit` directive. This technique leaves the recipe to describe just the things that are specific to the software being built. A `base` class exists that is implicitly inherited by all recipes and provides the functionality that most recipes typically need.

The remainder of this section presents information useful when working with recipes.

10.6.1. Finding Logs and Work Files¶

After the first run of the `devtool build` command, recipes that were previously created using the `devtool add` command or whose sources were modified using the `devtool modify` command contain symbolic links created within the source tree:

- `oe-logs`: This link points to the directory in which log files and run scripts for each build step are created.
- `oe-workdir`: This link points to the temporary work area for the recipe. The following locations under `oe-workdir` are particularly useful:
 - `image/`: Contains all of the files installed during the `do_install` stage. Within a recipe, this directory is referred to by the expression `${D}`.
 - `sysroot-destdir/`: Contains a subset of files installed within `do_install` that have been put into the shared sysroot. For more information, see the "[Sharing Files Between Recipes](#)" section.
 - `packages-split/`: Contains subdirectories for each package produced by the recipe. For more information, see the "[Packaging](#)" section.

You can use these links to get more information on what is happening at each build step.

10.6.2. Setting Configure Arguments¶

If the software your recipe is building uses GNU autoconf, then a fixed set of arguments is passed to it to enable cross-compilation plus any extras specified by `EXTRA_OECONF` or `PACKAGECONFIG_CONFARGS` set within the recipe. If you wish to pass additional options, add them to `EXTRA_OECONF` or `PACKAGECONFIG_CONFARGS`. Other supported build tools have similar variables (e.g. `EXTRA_OECMAKE` for CMake, `EXTRA_OESCONS` for Scons, and so forth). If you need to pass anything on the make command line, you can use `EXTRA_OEMAKE` or the `PACKAGECONFIG_CONFARGS` variables to do so.

You can use the `devtool configure-help` command to help you set the arguments listed in the previous paragraph. The command determines the exact options being passed, and shows them to you along with any custom arguments specified through `EXTRA_OECONF` or `PACKAGECONFIG_CONFARGS`. If applicable, the command also shows you the output of the configure script's "--help" option as a reference.

10.6.3. Sharing Files Between Recipes¶

Recipes often need to use files provided by other recipes on the [build host](#). For example, an application linking to a common library needs access to the library itself and its associated headers. The way this access is accomplished within the extensible SDK is through the sysroot. One sysroot exists per "machine" for which the SDK is being built. In practical terms, this means a sysroot exists for the target machine, and a sysroot exists for the build host.

Recipes should never write files directly into the sysroot. Instead, files should be installed into standard locations during the `do_install` task within the `${D}` directory. A subset of these files automatically goes into the sysroot. The reason for this limitation is that almost all files that go into the sysroot are cataloged in manifests in order to ensure they can be removed later when a recipe is modified or removed. Thus, the sysroot is able to remain free from stale files.

10.6.4. Packaging¶

Packaging is not always particularly relevant within the extensible SDK. However, if you examine how build output gets into the final image on the target device, it is important to understand packaging because the contents of the image are expressed in terms of packages and not recipes.

During the `do_package` task, files installed during the `do_install` task are split into one main package, which is almost always named the same as the recipe, and into several other packages. This separation exists because not all of those installed files are useful in every image. For example, you probably do not need any of the documentation installed in a production image. Consequently,

for each recipe the documentation files are separated into a `-doc` package. Recipes that package software containing optional modules or plugins might undergo additional package splitting as well.

After building a recipe, you can see where files have gone by looking in the `oe-workdir/packages-split` directory, which contains a subdirectory for each package. Apart from some advanced cases, the `PACKAGES` and `FILES` variables controls splitting. The `PACKAGES` variable lists all of the packages to be produced, while the `FILES` variable specifies which files to include in each package by using an override to specify the package. For example, `FILES_${PN}` specifies the files to go into the main package (i.e. the main package has the same name as the recipe and `${PN}` evaluates to the recipe name). The order of the `PACKAGES` value is significant. For each installed file, the first package whose `FILES` value matches the file is the package into which the file goes. Defaults exist for both the `PACKAGES` and `FILES` variables. Consequently, you might find you do not even need to set these variables in your recipe unless the software the recipe is building installs files into non-standard locations.

10.7. Restoring the Target Device to its Original State¶

If you use the `devtool deploy-target` command to write a recipe's build output to the target, and you are working on an existing component of the system, then you might find yourself in a situation where you need to restore the original files that existed prior to running the `devtool deploy-target` command. Because the `devtool deploy-target` command backs up any files it overwrites, you can use the `devtool undeploy-target` command to restore those files and remove any other files the recipe deployed. Consider the following example:

```
$ devtool undeploy-target lighttpd root@192.168.7.2
```

If you have deployed multiple applications, you can remove them all using the `-a` option thus restoring the target device to its original state:

```
$ devtool undeploy-target -a root@192.168.7.2
```

Information about files deployed to the target as well as any backed up files are stored on the target itself. This storage, of course, requires some additional space on the target machine.

Note

The `devtool deploy-target` and `devtool undeploy-target` commands do not currently interact with any package management system on the target device (e.g. RPM or OPKG). Consequently, you should not intermingle `devtool deploy-target` and package manager operations on the target device. Doing so could result in a conflicting set of files.

10.8. Installing Additional Items Into the Extensible SDK¶

Out of the box the extensible SDK typically only comes with a small number of tools and libraries. A minimal SDK starts mostly empty and is populated on-demand. Sometimes you must explicitly install extra items into the SDK. If you need these extra items, you can first search for the items using the `devtool search` command. For example, suppose you need to link to libGL but you are not sure which recipe provides libGL. You can use the following command to find out:

```
$ devtool search libGL
mesa                A free implementation of the OpenGL API
```

Once you know the recipe (i.e. `mesa` in this example), you can install it:

```
$ devtool sdk-install mesa
```

By default, the `devtool sdk-install` command assumes the item is available in pre-built form from your SDK provider. If the item is not available and it is acceptable to build the item from source, you can add the `-s` option as follows:

```
$ devtool sdk-install -s mesa
```

It is important to remember that building the item from source takes significantly longer than installing the pre-built artifact. Also, if no recipe exists for the item you want to add to the SDK, you must instead add the item using the `devtool add` command.

10.9. Applying Updates to an Installed Extensible SDK¶

If you are working with an installed extensible SDK that gets occasionally updated (e.g. a third-party SDK), then you will need to manually "pull down" the updates into the installed SDK.

To update your installed SDK, use `devtool` as follows:

```
$ devtool sdk-update
```

The previous command assumes your SDK provider has set the default update URL for you through the `SDK_UPDATE_URL` variable as described in the "Providing Updates to the Extensible SDK After Installation" section. If the SDK provider has not set that default URL, you need to specify it yourself in the command as follows:

```
$ devtool sdk-update path_to_update_directory
```

Note

The URL needs to point specifically to a published SDK and not to an SDK installer that you would download and install.

10.10. Creating a Derivative SDK With Additional Components¶

You might need to produce an SDK that contains your own custom libraries. A good example would be if you were a vendor with customers that use your SDK to build their own platform-specific software and those customers need an SDK that has custom libraries. In such a case, you can produce a derivative SDK based on the currently installed SDK fairly easily by following these steps:

1. If necessary, install an extensible SDK that you want to use as a base for your derivative SDK.
2. Source the environment script for the SDK.
3. Add the extra libraries or other components you want by using the `devtool add` command.
4. Run the `devtool build-sdk` command.

The previous steps take the recipes added to the workspace and construct a new SDK installer that contains those recipes and the resulting binary artifacts. The recipes go into their own separate layer in the constructed derivative SDK, which leaves the workspace clean and ready for users to add their own recipes.

Chapter 11. Using the Standard SDK¶

[11.1. Why use the Standard SDK and What is in It?](#)

[11.2. Installing the SDK](#)

[11.3. Running the SDK Environment Setup Script](#)

This chapter describes the standard SDK and how to install it. Information includes unique installation and setup aspects for the standard SDK.

Note

For a side-by-side comparison of main features supported for a standard SDK as compared to an extensible SDK, see the "[Introduction](#)" section.

You can use a standard SDK to work on Makefile and Autotools-based projects. See the "[Using the SDK Toolchain Directly](#)," chapter for more information.

11.1. Why use the Standard SDK and What is in It?¶

The Standard SDK provides a cross-development toolchain and libraries tailored to the contents of a specific image. You would use the Standard SDK if you want a more traditional toolchain experience as compared to the extensible SDK, which provides an internal build system and the `devtool` functionality.

The installed Standard SDK consists of several files and directories. Basically, it contains an SDK environment setup script, some configuration files, and host and target root filesystems to support usage. You can see the directory structure in the "[Installed Standard SDK Directory Structure](#)" section.

11.2. Installing the SDK¶

The first thing you need to do is install the SDK on your [Build Host](#) by running the `*.sh` installation script.

You can download a tarball installer, which includes the pre-built toolchain, the `runqemu` script, and support files from the appropriate [toolchain](#) directory within the Index of Releases. Toolchains are available for several 32-bit and 64-bit architectures with the `x86_64` directories, respectively. The toolchains the Yocto Project provides are based off the `core-image-sato` and `core-image-minimal` images and contain libraries appropriate for developing against that image.

The names of the tarball installer scripts are such that a string representing the host system appears first in the filename and then is immediately followed by a string representing the target architecture.

```
poky-glibc-host_system-image_type-arch-toolchain-release_version.sh
```

Where:

host_system is a string representing your development system:

i686 or x86_64.

image_type is the image for which the SDK was built:

core-image-minimal or core-image-sato.

arch is a string representing the tuned target architecture:

aarch64, armv5e, core2-64, i586, mips32r2, mips64, ppc7400, or cortexa8hf-neon.

release_version is a string representing the release number of the Yocto Project:

3.0.1, 3.0.1+snapshot

For example, the following SDK installer is for a 64-bit development host system and a i586-tuned target architecture based off the SDK for `core-image-sato` and using the current 3.0.1 snapshot:

poky-glibc-x86_64-core-image-sato-i586-toolchain-3.0.1.sh

Note

As an alternative to downloading an SDK, you can build the SDK installer. For information on building the installer, see the "[Building an SDK Installer](#)" section.

The SDK and toolchains are self-contained and by default are installed into the `poky_sdk` folder in your home directory. You can choose to install the extensible SDK in any location when you run the installer. However, because files need to be written under that directory during the normal course of operation, the location you choose for installation must be writable for whichever users need to use the SDK.

The following command shows how to run the installer given a toolchain tarball for a 64-bit x86 development host system and a 64-bit x86 target architecture. The example assumes the SDK installer is located in `~/Downloads/` and has execution rights.

Note

If you do not have write permissions for the directory into which you are installing the SDK, the installer notifies you and exits. For that case, set up the proper permissions in the directory and run the installer again.

```
$ ./Downloads/poky-glibc-x86_64-core-image-sato-i586-toolchain-3.0.1.sh
Poky (Yocto Project Reference Distro) SDK installer version 3.0.1
=====
Enter target directory for SDK (default: /opt/poky/3.0.1):

You are about to install the SDK to "/opt/poky/3.0.1". Proceed [Y/n]? Y
Extracting SDK.....done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.
$ . /opt/poky/3.0.1/environment-setup-i586-poky-linux
```

Again, reference the "[Installed Standard SDK Directory Structure](#)" section for more details on the resulting directory structure of the installed SDK.

11.3. Running the SDK Environment Setup Script¶

Once you have the SDK installed, you must run the SDK environment setup script before you can actually use the SDK. This setup script resides in the directory you chose when you installed the SDK, which is either the default `/opt/poky/3.0.1` directory or the directory you chose during installation.

Before running the script, be sure it is the one that matches the architecture for which you are developing. Environment setup scripts begin with the string "environment-setup" and include as part of their name the tuned target architecture. As an example, the following commands set the working directory to where the SDK was installed and then source the environment setup script. In this example, the setup script is for an IA-based target machine using i586 tuning:

```
$ source /opt/poky/3.0.1/environment-setup-i586-poky-linux
```

When you run the setup script, the same environment variables are defined as are when you run the setup script for an extensible SDK. See the "[Running the Extensible SDK Environment Setup Script](#)" section for more information.

Chapter 12. Using the SDK Toolchain Directly¶

[12.1. Autotools-Based Projects](#)

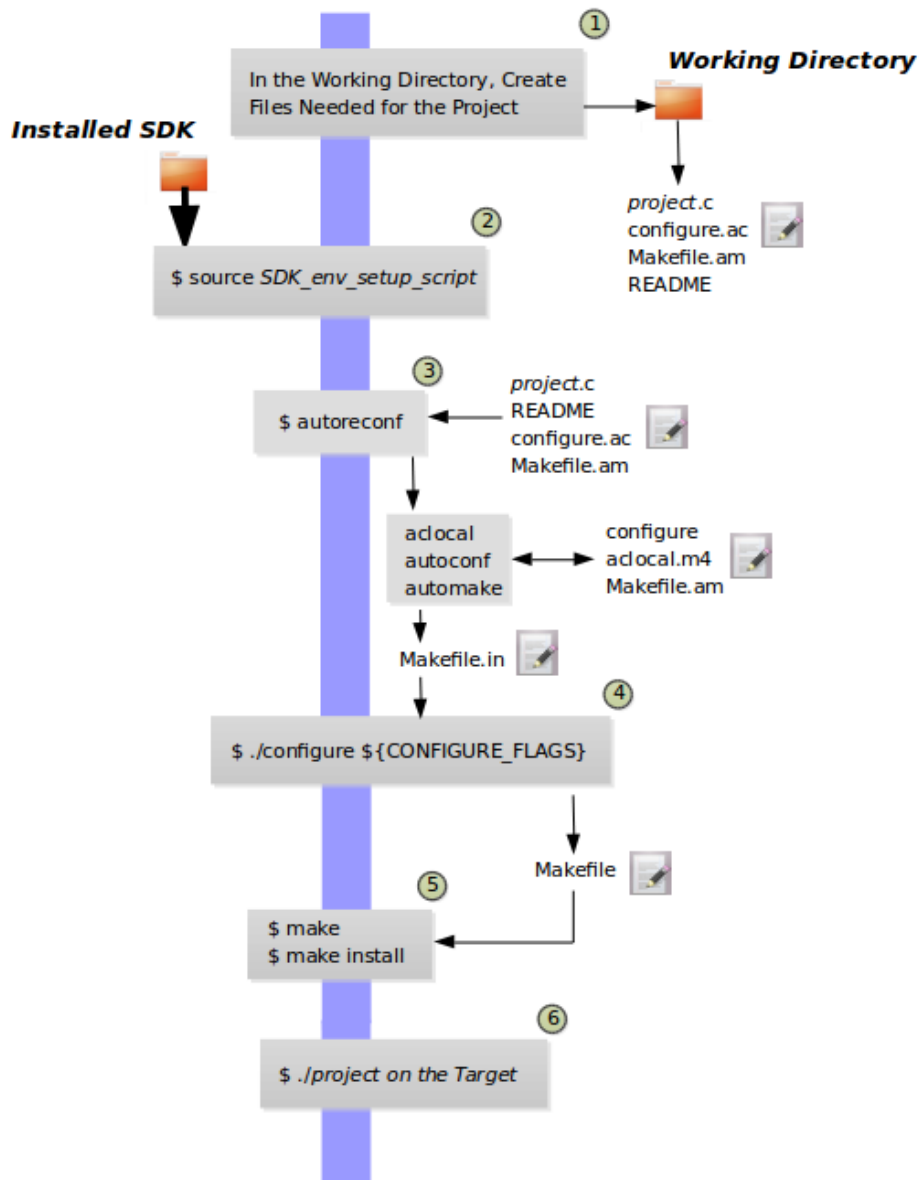
[12.2. Makefile-Based Projects](#)

You can use the SDK toolchain directly with Makefile and Autotools-based projects.

12.1. Autotools-Based Projects¶

Once you have a suitable [cross-development toolchain](#) installed, it is very easy to develop a project using the [GNU Autotools-based workflow](#), which is outside of the [OpenEmbedded build system](#).

The following figure presents a simple Autotools workflow.



Follow these steps to create a simple Autotools-based "Hello World" project:

Note

For more information on the GNU Autotools workflow, see the same example on the [GNOME Developer](#) site.

1. **Create a Working Directory and Populate It:** Create a clean directory for your project and then make that directory your working location.

```
$ mkdir $HOME/helloworld
$ cd $HOME/helloworld
```

After setting up the directory, populate it with files needed for the flow. You need a project source file, a file to help with configuration, and a file to help create the Makefile, and a README file: `hello.c`, `configure.ac`, `Makefile.am`, and `README`, respectively.

Use the following command to create an empty README file, which is required by GNU Coding Standards:

```
$ touch README
```

Create the remaining three files as follows:

- **hello.c:**

```
#include <stdio.h>

main()
{
    printf("Hello World!\n");
}
```

- **configure.ac:**

```
AC_INIT(hello,0.1)
AM_INIT_AUTOMAKE([foreign])
```

```
AC_PROG_CC
AC_CONFIG_FILES(Makefile)
AC_OUTPUT
```

• **Makefile.am:**

```
bin_PROGRAMS = hello
hello_SOURCES = hello.c
```

2. **Source the Cross-Toolchain Environment Setup File:** As described earlier in the manual, installing the cross-toolchain creates a cross-toolchain environment setup script in the directory that the SDK was installed. Before you can use the tools to develop your project, you must source this setup script. The script begins with the string "environment-setup" and contains the machine architecture, which is followed by the string "poky-linux". For this example, the command sources a script from the default SDK installation directory that uses the 32-bit Intel x86 Architecture and the Zeus Yocto Project release:

```
$ source /opt/poky/3.0.1/environment-setup-i586-poky-linux
```

3. **Create the configure Script:** Use the `autoreconf` command to generate the `configure` script.

```
$ autoreconf
```

The `autoreconf` tool takes care of running the other Autotools such as `aclocal`, `autoconf`, and `automake`.

Note

If you get errors from `configure.ac`, which `autoreconf` runs, that indicate missing files, you can use the `-i` option, which ensures missing auxiliary files are copied to the build host.

4. **Cross-Compile the Project:** This command compiles the project using the cross-compiler. The `CONFIGURE_FLAGS` environment variable provides the minimal arguments for GNU `configure`:

```
$ ./configure ${CONFIGURE_FLAGS}
```

For an Autotools-based project, you can use the cross-toolchain by just passing the appropriate host option to `configure.sh`. The host option you use is derived from the name of the environment setup script found in the directory in which you installed the cross-toolchain. For example, the host option for an ARM-based target that uses the GNU EABI is `armv5te-poky-linux-gnueabi`. You will notice that the name of the script is `environment-setup-armv5te-poky-linux-gnueabi`. Thus, the following command works to update your project and rebuild it using the appropriate cross-toolchain tools:

```
$ ./configure --host=armv5te-poky-linux-gnueabi --with-libtool-sysroot=sysroot_dir
```

5. **Make and Install the Project:** These two commands generate and install the project into the destination directory:

```
$ make
$ make install DESTDIR=./tmp
```

Note

To learn about environment variables established when you run the cross-toolchain environment setup script and how they are used or overridden when the Makefile, see the "[Makefile-Based Projects](#)" section.

This next command is a simple way to verify the installation of your project. Running the command prints the architecture on which the binary file can run. This architecture should be the same architecture that the installed cross-toolchain supports.

```
$ file ./tmp/usr/local/bin/hello
```

6. **Execute Your Project:** To execute the project, you would need to run it on your target hardware. If your target hardware happens to be your build host, you could run the project as follows:

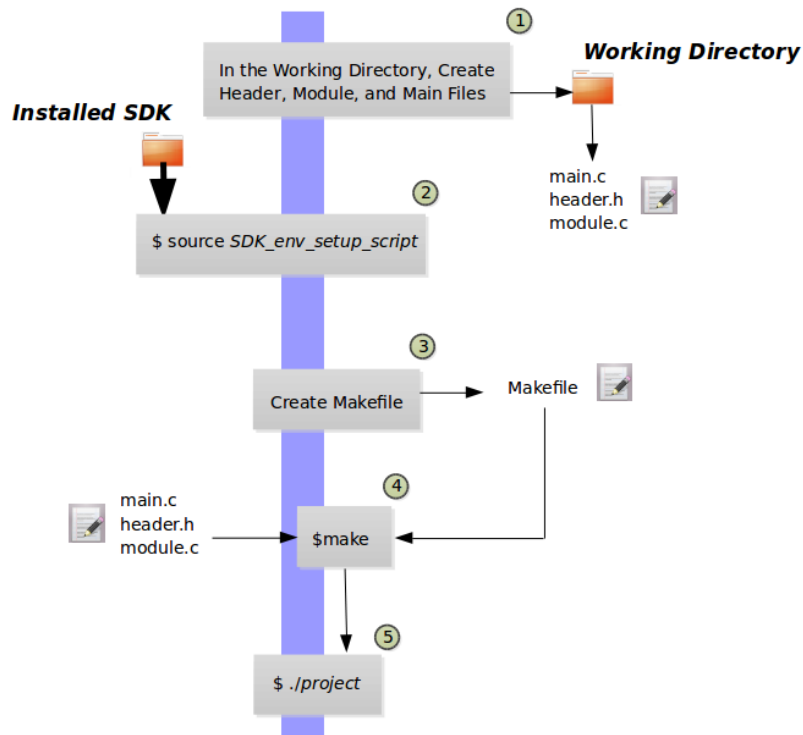
```
$ ./tmp/usr/local/bin/hello
```

As expected, the project displays the "Hello World!" message.

12.2. Makefile-Based Projects¶

Simple Makefile-based projects use and interact with the cross-toolchain environment variables established when you run the cross-toolchain environment setup script. The environment variables are subject to general `make` rules.

This section presents a simple Makefile development flow and provides an example that lets you see how you can use cross-toolchain environment variables and Makefile variables during development.



The main point of this section is to explain the following three cases regarding variable behavior:

- **Case 1 - No Variables Set in the Makefile Map to Equivalent Environment Variables Set in the SDK Setup Script:** Because matching variables are not specifically set in the `Makefile`, the variables retain their values based on the environment setup script.
- **Case 2 - Variables Are Set in the Makefile that Map to Equivalent Environment Variables from the SDK Setup Script:** Specifically setting matching variables in the `Makefile` during the build results in the environment settings of the variables being overwritten. In this case, the variables you set in the `Makefile` are used.
- **Case 3 - Variables Are Set Using the Command Line that Map to Equivalent Environment Variables from the SDK Setup Script:** Executing the `Makefile` from the command line results in the environment variables being overwritten. In this case, the command-line content is used.

Note

Regardless of how you set your variables, if you use the `-e` option with `make`, the variables from the SDK setup script take precedence:

```
$ make -e target
```

The remainder of this section presents a simple `Makefile` example that demonstrates these variable behaviors.

In a new shell environment variables are not established for the SDK until you run the setup script. For example, the following commands show a null value for the compiler variable (i.e. `CC`).

```
$ echo ${CC}
$
```

Running the SDK setup script for a 64-bit build host and an i586-tuned target architecture for a `core-image-sato` image using the current 3.0.1 Yocto Project release and then echoing that variable shows the value established through the script:

```
$ source /opt/poky/3.0.1/environment-setup-i586-poky-linux
$ echo ${CC}
i586-poky-linux-gcc -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux
```

To illustrate variable use, work through this simple "Hello World!" example:

1. **Create a Working Directory and Populate It:** Create a clean directory for your project and then make that directory your working location.

```
$ mkdir $HOME/helloworld
$ cd $HOME/helloworld
```

After setting up the directory, populate it with files needed for the flow. You need a `main.c` file from which you call your function, a `module.h` file to contain headers, and a `module.c` that defines your function.

Create the three files as follows:

- **main.c:**


```
#include "module.h"
void sample_func();
int main()
{
    sample_func();
    return 0;
}
```

- **module.h:**

```
#include <stdio.h>
void sample_func();
```

- **module.c:**

```
#include "module.h"
void sample_func()
{
    printf("Hello World!");
    printf("\n");
}
```

2. **Source the Cross-Toolchain Environment Setup File:** As described earlier in the manual, installing the cross-toolchain creates a cross-toolchain environment setup script in the directory that the SDK was installed. Before you can use the tools to develop your project, you must source this setup script. The script begins with the string "environment-setup" and contains the machine architecture, which is followed by the string "poky-linux". For this example, the command sources a script from the default SDK installation directory that uses the 32-bit Intel x86 Architecture and the Zeus Yocto Project release:

```
$ source /opt/poky/3.0.1/environment-setup-i586-poky-linux
```

3. **Create the Makefile:** For this example, the Makefile contains two lines that can be used to set the `CC` variable. One line is identical to the value that is set when you run the SDK environment setup script, and the other line sets `CC` to "gcc", the default GNU compiler on the build host:

```
# CC=i586-poky-linux-gcc -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux
# CC="gcc"
all: main.o module.o
    ${CC} main.o module.o -o target_bin
main.o: main.c module.h
    ${CC} -I . -c main.c
module.o: module.c module.h
    ${CC} -I . -c module.c
clean:
    rm -rf *.o
    rm target_bin
```

4. **Make the Project:** Use the `make` command to create the binary output file. Because variables are commented out in the Makefile, the value used for `CC` is the value set when the SDK environment setup file was run:

```
$ make
i586-poky-linux-gcc -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c main.c
i586-poky-linux-gcc -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c module.c
i586-poky-linux-gcc -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux main.o module.o -o target_b:
```

From the results of the previous command, you can see that the compiler used was the compiler established through the `CC` variable defined in the setup script.

You can override the `CC` environment variable with the same variable as set from the Makefile by uncommenting the line in the Makefile and running `make` again.

```
$ make clean
rm -rf *.o
rm target_bin
#
# Edit the Makefile by uncommenting the line that sets CC to "gcc"
#
$ make
gcc -I . -c main.c
gcc -I . -c module.c
gcc main.o module.o -o target_bin
```

As shown in the previous example, the cross-toolchain compiler is not used. Rather, the default compiler is used.

This next case shows how to override a variable by providing the variable as part of the command line. Go into the Makefile and re-insert the comment character so that running `make` uses the established SDK compiler. However, when you run `make`, use a command-line argument to set `CC` to "gcc":

```
$ make clean
rm -rf *.o
rm target_bin
#
# Edit the Makefile to comment out the line setting CC to "gcc"
#
$ make
i586-poky-linux-gcc -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c main.c
i586-poky-linux-gcc -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c module.c
i586-poky-linux-gcc -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux main.o module.o -o target_b:
$ make clean
rm -rf *.o
rm target_bin
$ make CC="gcc"
gcc -I . -c main.c
gcc -I . -c module.c
```

```
gcc main.o module.o -o target_bin
```

In the previous case, the command-line argument overrides the SDK environment variable.

In this last case, edit Makefile again to use the "gcc" compiler but then use the "-e" option on the `make` command line:

```
$ make clean
rm -rf *.o
rm target_bin
#
# Edit the Makefile to use "gcc"
#
$ make
gcc -I . -c main.c
gcc -I . -c module.c
gcc main.o module.o -o target_bin
$ make clean
rm -rf *.o
rm target_bin
$ make -e
i586-poky-linux-gcc -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c main.c
i586-poky-linux-gcc -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c module.c
i586-poky-linux-gcc -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux main.o module.o -o target_bin
```

In the previous case, the "-e" option forces `make` to use the SDK environment variables regardless of the values in the Makefile.

5. **Execute Your Project:** To execute the project (i.e. `target_bin`), use the following command:

```
$ ./target_bin
Hello World!
```

Note

If you used the cross-toolchain compiler to build `target_bin` and your build host differs in architecture from that of the target machine, you need to run your project on the target device.

As expected, the project displays the "Hello World!" message.

Appendix A. Obtaining the SDK¶

[A.1. Locating Pre-Built SDK Installers](#)

[A.2. Building an SDK Installer](#)

[A.3. Extracting the Root Filesystem](#)

[A.4. Installed Standard SDK Directory Structure](#)

[A.5. Installed Extensible SDK Directory Structure](#)

A.1. Locating Pre-Built SDK Installers¶

You can use existing, pre-built toolchains by locating and running an SDK installer script that ships with the Yocto Project. Using this method, you select and download an architecture-specific SDK installer and then run the script to hand-install the toolchain.

Follow these steps to locate and hand-install the toolchain:

1. **Go to the Installers Directory:** Go to <http://downloads.yoctoproject.org/releases/yocto/yocto-3.0.1/toolchain/>
2. **Open the Folder for Your Build Host:** Open the folder that matches your [build host](#) (i.e. `i686` for 32-bit machines or `x86_64` for 64-bit machines).
3. **Locate and Download the SDK Installer:** You need to find and download the installer appropriate for your build host, target hardware, and image type.

The installer files (`*.sh`) follow this naming convention:

```
poky-glibc-host_system-core-image-type-arch-toolchain[-ext]-release.sh
```

Where:

`host_system` is a string representing your development system:
"i686" or "x86_64"

`type` is a string representing the image:
"sato" or "minimal"

`arch` is a string representing the target architecture:
"aarch64", "armv5e", "core2-64", "cortexa8hf-neon", "i586", "mips32r2",
"mips64", or "ppc7400"

`release` is the version of Yocto Project.

NOTE:

The standard SDK installer does not have the "-ext" string as part of the filename.

The toolchains provided by the Yocto Project are based off of the `core-image-sato` and `core-image-minimal` images and contain libraries appropriate for developing against those images.

For example, if your build host is a 64-bit x86 system and you need an extended SDK for a 64-bit core2 target, go into the `x86_64` folder and download the following installer:

```
poky-glibc-x86_64-core-image-sato-core2-64-toolchain-ext-3.0.1.sh
```

4. **Run the Installer:** Be sure you have execution privileges and run the installer. Following is an example from the Downloads directory:

```
$ ~/Downloads/poky-glibc-x86_64-core-image-sato-core2-64-toolchain-ext-3.0.1.sh
```

During execution of the script, you choose the root location for the toolchain. See the "[Installed Standard SDK Directory Structure](#)" section and the "[Installed Extensible SDK Directory Structure](#)" section for more information.

A.2. Building an SDK Installer¶

As an alternative to locating and downloading an SDK installer, you can build the SDK installer. Follow these steps:

1. **Set Up the Build Environment:** Be sure you are set up to use BitBake in a shell. See the "[Preparing the Build Host](#)" section in the Yocto Project Development Tasks Manual for information on how to get a build host ready that is either a native Linux machine or a machine that uses CROPS.
2. **Clone the poky Repository:** You need to have a local copy of the Yocto Project [Source Directory](#) (i.e. a local poky repository). See the "[Cloning the poky Repository](#)" and possibly the "[Checking Out by Branch in Poky](#)" and "[Checking Out by Tag in Poky](#)" sections all in the Yocto Project Development Tasks Manual for information on how to clone the poky repository and check out the appropriate branch for your work.
3. **Initialize the Build Environment:** While in the root directory of the Source Directory (i.e. poky), run the `oe-init-build-env` environment setup script to define the OpenEmbedded build environment on your build host.

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

Among other things, the script creates the [Build Directory](#), which is `build` in this case and is located in the Source Directory. After the script runs, your current working directory is set to the `build` directory.

4. **Make Sure You Are Building an Installer for the Correct Machine:** Check to be sure that your `MACHINE` variable in the `local.conf` file in your Build Directory matches the architecture for which you are building.
5. **Make Sure Your SDK Machine is Correctly Set:** If you are building a toolchain designed to run on an architecture that differs from your current development host machine (i.e. the build host), be sure that the `SDKMACHINE` variable in the `local.conf` file in your Build Directory is correctly set.

Note

If you are building an SDK installer for the Extensible SDK, the `SDKMACHINE` value must be set for the architecture of the machine you are using to build the installer. If `SDKMACHINE` is not set appropriately, the build fails and provides an error message similar to the following:

```
The extensible SDK can currently only be built for the same architecture as the machine being set to i686 (likely via setting SDKMACHINE) which is different from the architecture of the Unable to continue.
```



6. **Build the SDK Installer:** To build the SDK installer for a standard SDK and populate the SDK image, use the following command form. Be sure to replace `image` with an image (e.g. "core-image-sato"):

```
$ bitbake image -c populate_sdk
```

You can do the same for the extensible SDK using this command form:

```
$ bitbake image -c populate_sdk_ext
```

These commands produce an SDK installer that contains the sysroot that matches your target root filesystem.

When the `bitbake` command completes, the SDK installer will be in `tmp/depot/sdk` in the Build Directory.

Notes

- By default, the previous BitBake command does not build static binaries. If you want to use the toolchain to build these types of libraries, you need to be sure your SDK has the appropriate static development libraries. Use the `TOOLCHAIN_TARGET_TASK` variable inside your `local.conf` file before building the SDK installer. Doing so ensures that the eventual SDK installation process installs the appropriate library packages as part of the SDK. Following is an example using `libc` static development libraries:

```
TOOLCHAIN_TARGET_TASK_append = " libc-staticdev"
```

7. **Run the Installer:** You can now run the SDK installer from `tmp/deploy/sdk` in the Build Directory. Following is an example:

```
$ cd ~/poky/build/tmp/deploy/sdk
$ ./poky-glibc-x86_64-core-image-sato-core2-64-toolchain-ext-3.0.1.sh
```

During execution of the script, you choose the root location for the toolchain. See the "[Installed Standard SDK Directory Structure](#)" section and the "[Installed Extensible SDK Directory Structure](#)" section for more information.

A.3. Extracting the Root Filesystem¶

After installing the toolchain, for some use cases you might need to separately extract a root filesystem:

- You want to boot the image using NFS.
- You want to use the root filesystem as the target sysroot.
- You want to develop your target application using the root filesystem as the target sysroot.

Follow these steps to extract the root filesystem:

1. **Locate and Download the Tarball for the Pre-Built Root Filesystem Image File:** You need to find and download the root filesystem image file that is appropriate for your target system. These files are kept in machine-specific folders in the [Index of Releases](#) in the "machines" directory.

The machine-specific folders of the "machines" directory contain tarballs (`*.tar.bz2`) for supported machines. These directories also contain flattened root filesystem image files (`*.ext4`), which you can use with QEMU directly.

The pre-built root filesystem image files follow these naming conventions:

```
core-image-profile-arch.tar.bz2
```

Where:

profile is the filesystem image's profile:
lsb, lsb-dev, lsb-sdk, minimal, minimal-dev, minimal-initramfs, sato, sato-dev, sato-sdk, sato-sdk-ptest. For information on these types of image profiles, see the "[Images](#)" chapter in the Yocto Project Reference Manual.

arch is a string representing the target architecture:
beaglebone-yocto, beaglebone-yocto-lsb, edgerouter, edgerouter-lsb, genericx86, genericx86-64, genericx86-64-lsb, genericx86-lsb, mpc8315e-rdb, mpc8315e-rdb-lsb, and qemu*.

The root filesystems provided by the Yocto Project are based off of the `core-image-sato` and `core-image-minimal` images.

For example, if you plan on using a BeagleBone device as your target hardware and your image is a `core-image-sato-sdk` image, you can download the following file:

```
core-image-sato-sdk-beaglebone-yocto.tar.bz2
```

2. **Initialize the Cross-Development Environment:** You must `source` the cross-development environment setup script to establish necessary environment variables.

This script is located in the top-level directory in which you installed the toolchain (e.g. `poky_sdk`).

Following is an example based on the toolchain installed in the "[Locating Pre-Built SDK Installers](#)" section:

```
$ source ~/poky_sdk/environment-setup-core2-64-poky-linux
```

3. **Extract the Root Filesystem:** Use the `runqemu-extract-sdk` command and provide the root filesystem image.

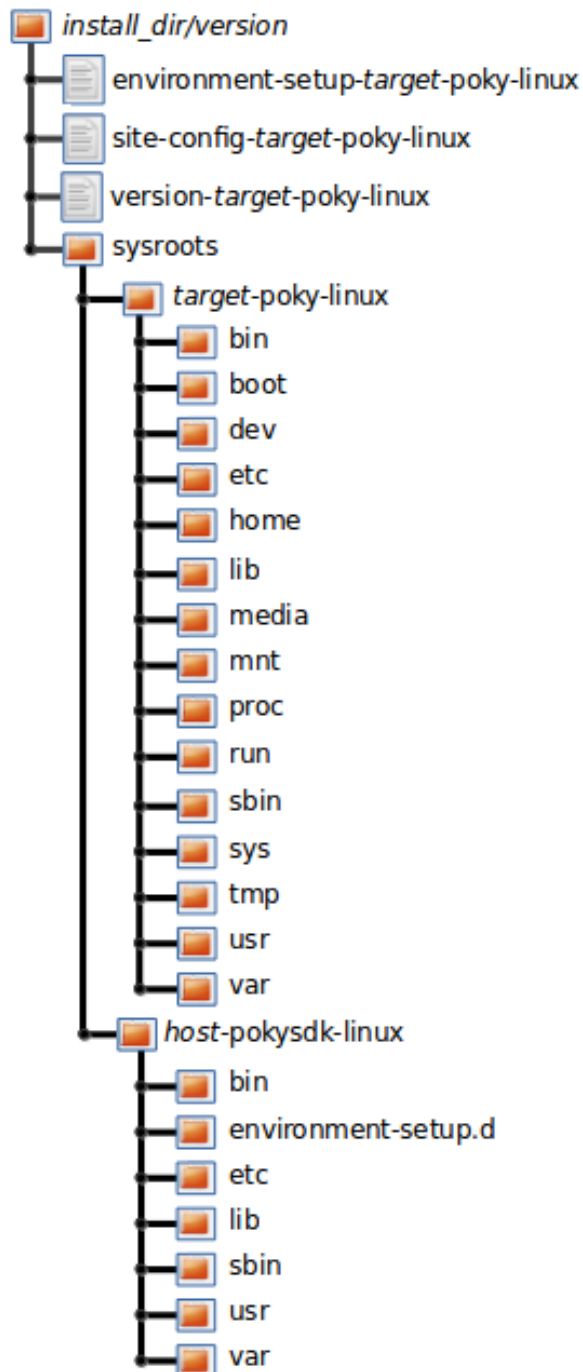
Following is an example command that extracts the root filesystem from a previously built root filesystem image that was downloaded from the [Index of Releases](#). This command extracts the root filesystem into the `core2-64-sato` directory:

```
$ runqemu-extract-sdk ~/Downloads/core-image-sato-sdk-beaglebone-yocto.tar.bz2 ~/beaglebone-sato
```

You could now point to the target sysroot at `beaglebone-sato`.

A.4. Installed Standard SDK Directory Structure¶

The following figure shows the resulting directory structure after you install the Standard SDK by running the `*.sh` SDK installation script:

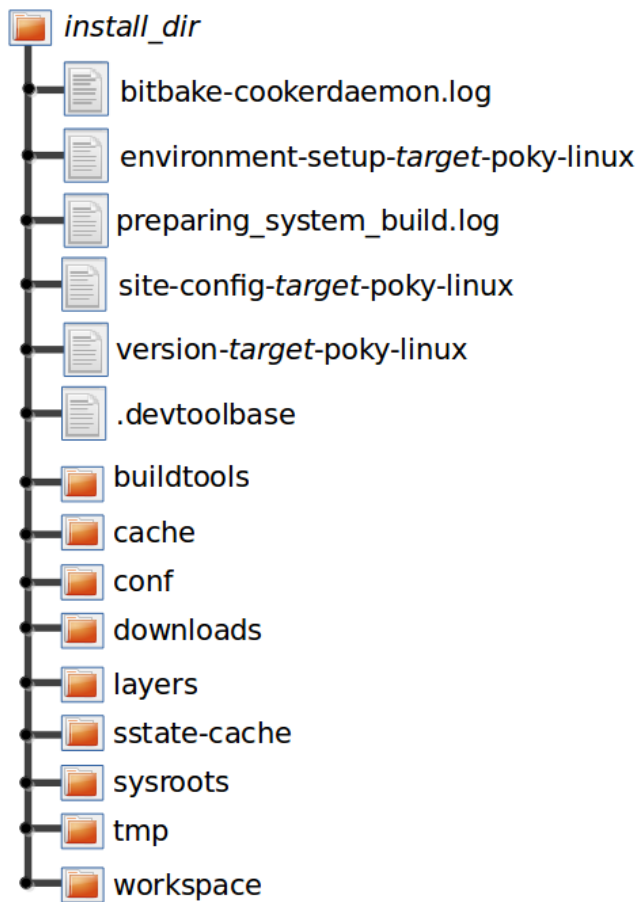


The installed SDK consists of an environment setup script for the SDK, a configuration file for the target, a version file for the target, and the root filesystem (`sysroots`) needed to develop objects for the target system.

Within the figure, italicized text is used to indicate replaceable portions of the file or directory name. For example, *install_dir/version* is the directory where the SDK is installed. By default, this directory is `/opt/poky/`. And, *version* represents the specific snapshot of the SDK (e.g. `3.0.1`). Furthermore, *target* represents the target architecture (e.g. `i586`) and *host* represents the development system's architecture (e.g. `x86_64`). Thus, the complete names of the two directories within the `sysroots` could be `i586-poky-linux` and `x86_64-pokysdk-linux` for the target and host, respectively.

A.5. Installed Extensible SDK Directory Structure¶

The following figure shows the resulting directory structure after you install the Extensible SDK by running the `*.sh` SDK installation script:



The installed directory structure for the extensible SDK is quite different than the installed structure for the standard SDK. The extensible SDK does not separate host and target parts in the same manner as does the standard SDK. The extensible SDK uses an embedded copy of the OpenEmbedded build system, which has its own sysroots.

Of note in the directory structure are an environment setup script for the SDK, a configuration file for the target, a version file for the target, and log files for the OpenEmbedded build system preparation script run by the installer and BitBake.

Within the figure, italicized text is used to indicate replaceable portions of the file or directory name. For example, *install_dir* is the directory where the SDK is installed, which is *poky_sdk* by default, and *target* represents the target architecture (e.g. *i586*).

Appendix B. Customizing the Extensible SDK¶

[B.1. Configuring the Extensible SDK](#)

[B.2. Adjusting the Extensible SDK to Suit Your Build Host's Setup](#)

[B.3. Changing the Extensible SDK Installer Title](#)

[B.4. Providing Updates to the Extensible SDK After Installation](#)

[B.5. Changing the Default SDK Installation Directory](#)

[B.6. Providing Additional Installable Extensible SDK Content](#)

[B.7. Minimizing the Size of the Extensible SDK Installer Download](#)

This appendix describes customizations you can apply to the extensible SDK.

B.1. Configuring the Extensible SDK¶

The extensible SDK primarily consists of a pre-configured copy of the OpenEmbedded build system from which it was produced. Thus, the SDK's configuration is derived using that build system and the filters shown in the following list. When these filters are present, the OpenEmbedded build system applies them against `local.conf` and `auto.conf`:

- Variables whose values start with "/" are excluded since the assumption is that those values are paths that are likely to be specific to the `build host`.
- Variables listed in `SDK_LOCAL_CONF_BLACKLIST` are excluded. These variables are not allowed through from the OpenEmbedded build system configuration into the extensible SDK configuration. Typically, these variables are specific to the machine on which the build system is running and could be problematic as part of the extensible SDK configuration.

For a list of the variables excluded by default, see the `SDK_LOCAL_CONF_BLACKLIST` in the glossary of the Yocto Project Reference Manual.

- Variables listed in `SDK_LOCAL_CONF_WHITELIST` are included. Including a variable in the value of `SDK_LOCAL_CONF_WHITELIST` overrides either of the previous two filters. The default value is blank.
- Classes inherited globally with `INHERIT` that are listed in `SDK_INHERIT_BLACKLIST` are disabled. Using `SDK_INHERIT_BLACKLIST` to disable these classes is the typical method to disable classes that are problematic or unnecessary in the SDK context. The default value blacklists the `buildhistory` and `icecc` classes.

Additionally, the contents of `conf/sdk-extra.conf`, when present, are appended to the end of `conf/local.conf` within the produced SDK, without any filtering. The `sdk-extra.conf` file is particularly useful if you want to set a variable value just for the SDK and not the OpenEmbedded build system used to create the SDK.

B.2. Adjusting the Extensible SDK to Suit Your Build Host's Setup¶

In most cases, the extensible SDK defaults should work with your [build host's](#) setup. However, some cases exist for which you might consider making adjustments:

- If your SDK configuration inherits additional classes using the `INHERIT` variable and you do not need or want those classes enabled in the SDK, you can blacklist them by adding them to the `SDK_INHERIT_BLACKLIST` variable as described in the fourth bullet of the previous section.

Note

The default value of `SDK_INHERIT_BLACKLIST` is set using the `"?="` operator. Consequently, you will need to either define the entire list by using the `"="` operator, or you will need to append a value using either `"_append"` or the `"+="` operator. You can learn more about these operators in the ["Basic Syntax"](#) section of the BitBake User Manual.

- If you have classes or recipes that add additional tasks to the standard build flow (i.e. the tasks execute as the recipe builds as opposed to being called explicitly), then you need to do one of the following:
 - After ensuring the tasks are [shared state](#) tasks (i.e. the output of the task is saved to and can be restored from the shared state cache) or ensuring the tasks are able to be produced quickly from a task that is a shared state task, add the task name to the value of `SDK_RECRDEP_TASKS`.
 - Disable the tasks if they are added by a class and you do not need the functionality the class provides in the extensible SDK. To disable the tasks, add the class to the `SDK_INHERIT_BLACKLIST` variable as described in the previous section.
- Generally, you want to have a shared state mirror set up so users of the SDK can add additional items to the SDK after installation without needing to build the items from source. See the ["Providing Additional Installable Extensible SDK Content"](#) section for information.
- If you want users of the SDK to be able to easily update the SDK, you need to set the `SDK_UPDATE_URL` variable. For more information, see the ["Providing Updates to the Extensible SDK After Installation"](#) section.
- If you have adjusted the list of files and directories that appear in `COREBASE` (other than layers that are enabled through `bblayers.conf`), then you must list these files in `COREBASE_FILES` so that the files are copied into the SDK.
- If your OpenEmbedded build system setup uses a different environment setup script other than `oe-init-build-env`, then you must set `OE_INIT_ENV_SCRIPT` to point to the environment setup script you use.

Note

You must also reflect this change in the value used for the `COREBASE_FILES` variable as previously described.

B.3. Changing the Extensible SDK Installer Title¶

You can change the displayed title for the SDK installer by setting the `SDK_TITLE` variable and then rebuilding the the SDK installer. For information on how to build an SDK installer, see the ["Building an SDK Installer"](#) section.

By default, this title is derived from `DISTRO_NAME` when it is set. If the `DISTRO_NAME` variable is not set, the title is derived from the `DISTRO` variable.

The `populate_sdk_base` class defines the default value of the `SDK_TITLE` variable as follows:

```
SDK_TITLE ??= "${@d.getVar('DISTRO_NAME') or d.getVar('DISTRO')}" SDK"
```

While several ways exist to change this variable, an efficient method is to set the variable in your distribution's configuration file. Doing so creates an SDK installer title that applies across your distribution. As an example, assume you have your own layer for your distribution named "meta-mydistro" and you are using the same type of file hierarchy as does the default "poky" distribution. If so, you could update the `SDK_TITLE` variable in the `~/meta-mydistro/conf/distro/mydistro.conf` file using the following form:

```
SDK_TITLE = "your_title"
```

B.4. Providing Updates to the Extensible SDK After Installation¶

When you make changes to your configuration or to the metadata and if you want those changes to be reflected in installed SDKs, you need to perform additional steps. These steps make it possible for anyone using the installed SDKs to update the installed SDKs by using the `devtool sdk-update` command:

1. Create a directory that can be shared over HTTP or HTTPS. You can do this by setting up a web server such as an [Apache HTTP Server](#) or [Nginx](#) server in the cloud to host the directory. This directory must contain the published SDK.

2. Set the `SDK_UPDATE_URL` variable to point to the corresponding HTTP or HTTPS URL. Setting this variable causes any SDK built to default to that URL and thus, the user does not have to pass the URL to the `devtool sdk-update` command as described in the "Applying Updates to an Installed Extensible SDK" section.
3. Build the extensible SDK normally (i.e., use the `bitbake -c populate_sdk_ext imagename` command).
4. Publish the SDK using the following command:

```
$ oe-publish-sdk some_path/sdk-installer.sh path_to_shared_http_directory
```

You must repeat this step each time you rebuild the SDK with changes that you want to make available through the update mechanism.

Completing the above steps allows users of the existing installed SDKs to simply run `devtool sdk-update` to retrieve and apply the latest updates. See the "Applying Updates to an Installed Extensible SDK" section for further information.

B.5. Changing the Default SDK Installation Directory¶

When you build the installer for the Extensible SDK, the default installation directory for the SDK is based on the `DISTRO` and `SDKEXTPATH` variables from within the `populate_sdk_base` class as follows:

```
SDKEXTPATH ??= "~/${d.getVar('DISTRO')}_sdk"
```

You can change this default installation directory by specifically setting the `SDKEXTPATH` variable.

While a number of ways exist through which you can set this variable, the method that makes the most sense is to set the variable in your distribution's configuration file. Doing so creates an SDK installer default directory that applies across your distribution. As an example, assume you have your own layer for your distribution named "meta-mydistro" and you are using the same type of file hierarchy as does the default "poky" distribution. If so, you could update the `SDKEXTPATH` variable in the `~/meta-mydistro/conf/distro/mydistro.conf` file using the following form:

```
SDKEXTPATH = "some_path_for_your_installed_sdk"
```

After building your installer, running it prompts the user for acceptance of the `some_path_for_your_installed_sdk` directory as the default location to install the Extensible SDK.

B.6. Providing Additional Installable Extensible SDK Content¶

If you want the users of an extensible SDK you build to be able to add items to the SDK without requiring the users to build the items from source, you need to do a number of things:

1. Ensure the additional items you want the user to be able to install are already built:
 - Build the items explicitly. You could use one or more "meta" recipes that depend on lists of other recipes.
 - Build the "world" target and set `EXCLUDE_FROM_WORLD_pn-recipeName` for the recipes you do not want built. See the `EXCLUDE_FROM_WORLD` variable for additional information.
2. Expose the `sstate-cache` directory produced by the build. Typically, you expose this directory by making it available through an [Apache HTTP Server](#) or [Nginx](#) server.
3. Set the appropriate configuration so that the produced SDK knows how to find the configuration. The variable you need to set is `SSTATE_MIRRORS`:

```
SSTATE_MIRRORS = "file://.* http://example.com/some_path/sstate-cache/PATH"
```

You can set the `SSTATE_MIRRORS` variable in two different places:

- If the mirror value you are setting is appropriate to be set for both the OpenEmbedded build system that is actually building the SDK and the SDK itself (i.e. the mirror is accessible in both places or it will fail quickly on the OpenEmbedded build system side, and its contents will not interfere with the build), then you can set the variable in your `local.conf` or custom distro configuration file. You can then "whitelist" the variable through to the SDK by adding the following:
- ```
SDK_LOCAL_CONF_WHITELIST = "SSTATE_MIRRORS"
```
- Alternatively, if you just want to set the `SSTATE_MIRRORS` variable's value for the SDK alone, create a `conf/sdk-extra.conf` file either in your [Build Directory](#) or within any layer and put your `SSTATE_MIRRORS` setting within that file.

### Note

This second option is the safest option should you have any doubts as to which method to use when setting `SSTATE_MIRRORS`.

## B.7. Minimizing the Size of the Extensible SDK Installer Download¶

By default, the extensible SDK bundles the shared state artifacts for everything needed to reconstruct the image for which the SDK was built. This bundling can lead to an SDK installer file that is a Gigabyte or more in size. If the size of this file causes a problem, you can build an SDK that has just enough in it to install and provide access to the `devtool` command by setting the following in your configuration:

```
SDK_EXT_TYPE = "minimal"
```

Setting `SDK_EXT_TYPE` to "minimal" produces an SDK installer that is around 35 Mbytes in size, which downloads and installs quickly. You need to realize, though, that the minimal installer does not install any libraries or tools out of the box. These libraries and tools must be installed either "on the fly" or through actions you perform using `devtool` or explicitly with the `devtool sdk-install` command.

In most cases, when building a minimal SDK you need to also enable bringing in the information on a wider range of packages produced by the system. Requiring this wider range of information is particularly true so that `devtool add` is able to effectively map dependencies it discovers in a source tree to the appropriate recipes. Additionally, the information enables the `devtool search` command to return useful results.

To facilitate this wider range of information, you would need to set the following:

```
SDK_INCLUDE_PKGDATA = "1"
```

See the `SDK_INCLUDE_PKGDATA` variable for additional information.

Setting the `SDK_INCLUDE_PKGDATA` variable as shown causes the "world" target to be built so that information for all of the recipes included within it are available. Having these recipes available increases build time significantly and increases the size of the SDK installer by 30-80 Mbytes depending on how many recipes are included in your configuration.

You can use `EXCLUDE_FROM_WORLD_pn-recipeName` for recipes you want to exclude. However, it is assumed that you would need to be building the "world" target if you want to provide additional items to the SDK. Consequently, building for "world" should not represent undue overhead in most cases.

## Note

If you set `SDK_EXT_TYPE` to "minimal", then providing a shared state mirror is mandatory so that items can be installed as needed. See the "[Providing Additional Installable Extensible SDK Content](#)" section for more information.

You can explicitly control whether or not to include the toolchain when you build an SDK by setting the `SDK_INCLUDE_TOOLCHAIN` variable to "1". In particular, it is useful to include the toolchain when you have set `SDK_EXT_TYPE` to "minimal", which by default, excludes the toolchain. Also, it is helpful if you are building a small SDK for use with an IDE or some other tool where you do not want to take extra steps to install a toolchain.

## Appendix C. Customizing the Standard SDK¶

[C.1. Adding Individual Packages to the Standard SDK](#)

[C.2. Adding API Documentation to the Standard SDK](#)

This appendix presents customizations you can apply to the standard SDK.

### C.1. Adding Individual Packages to the Standard SDK¶

When you build a standard SDK using the `bitbake -c populate_sdk`, a default set of packages is included in the resulting SDK. The `TOOLCHAIN_HOST_TASK` and `TOOLCHAIN_TARGET_TASK` variables control the set of packages adding to the SDK.

If you want to add individual packages to the toolchain that runs on the host, simply add those packages to the `TOOLCHAIN_HOST_TASK` variable. Similarly, if you want to add packages to the default set that is part of the toolchain that runs on the target, add the packages to the `TOOLCHAIN_TARGET_TASK` variable.

### C.2. Adding API Documentation to the Standard SDK¶

You can include API documentation as well as any other documentation provided by recipes with the standard SDK by adding "api-documentation" to the `DISTRO_FEATURES` variable:

```
DISTRO_FEATURES_append = " api-documentation"
```

Setting this variable as shown here causes the OpenEmbedded build system to build the documentation and then include it in the standard SDK.

# Yocto Project Board Support Package (BSP) Developer's Guide



## Chapter 13. Board Support Packages (BSP) - Developer's Guide¶

- [13.1. BSP Layers](#)
- [13.2. Preparing Your Build Host to Work With BSP Layers](#)
- [13.3. Example Filesystem Layout](#)
  - [13.3.1. License Files](#)
  - [13.3.2. README File](#)
  - [13.3.3. README.sources File](#)
  - [13.3.4. Pre-built User Binaries](#)
  - [13.3.5. Layer Configuration File](#)
  - [13.3.6. Hardware Configuration Options](#)
  - [13.3.7. Miscellaneous BSP-Specific Recipe Files](#)
  - [13.3.8. Display Support Files](#)
  - [13.3.9. Linux Kernel Configuration](#)
- [13.4. Developing a Board Support Package \(BSP\)](#)
- [13.5. Requirements and Recommendations for Released BSPs](#)
  - [13.5.1. Released BSP Requirements](#)
  - [13.5.2. Released BSP Recommendations](#)
- [13.6. Customizing a Recipe for a BSP](#)
- [13.7. BSP Licensing Considerations](#)
- [13.8. Creating a new BSP Layer Using the `bitbake-layers` Script](#)
  - [13.8.1. BSP Layer Configuration Example](#)
  - [13.8.2. BSP Machine Configuration Example](#)
  - [13.8.3. BSP Kernel Recipe Example](#)

A Board Support Package (BSP) is a collection of information that defines how to support a particular hardware device, set of devices, or hardware platform. The BSP includes information about the hardware features present on the device and kernel configuration information along with any additional hardware drivers required. The BSP also lists any additional software components required in addition to a generic Linux software stack for both essential and optional platform features.

This guide presents information about BSP Layers, defines a structure for components so that BSPs follow a commonly understood layout, discusses how to customize a recipe for a BSP, addresses BSP licensing, and provides information that shows you how to create a [BSP Layer](#) using the [bitbake-layers](#) tool.

### 13.1. BSP Layers¶

A BSP consists of a file structure inside a base directory. Collectively, you can think of the base directory, its file structure, and the contents as a BSP Layer. Although not a strict requirement, BSP layers in the Yocto Project use the following well-established naming convention:

```
meta-bsp_root_name
```

The string "meta-" is prepended to the machine or platform name, which is `bsp_root_name` in the above form.

#### Tip

Because the BSP layer naming convention is well-established, it is advisable to follow it when creating layers. Technically speaking, a BSP layer name does not need to start with `meta-`. However, various scripts and tools in the Yocto Project development environment assume this convention.

To help understand the BSP layer concept, consider the BSPs that the Yocto Project supports and provides with each release. You can see the layers in the [Yocto Project Source Repositories](#) through a web interface at <http://git.yoctoproject.org>. If you go to that interface, you will find a list of repositories under "Yocto Metadata Layers".

#### Note

Layers that are no longer actively supported as part of the Yocto Project appear under the heading "Yocto Metadata Layer Archive."

Each repository is a BSP layer supported by the Yocto Project (e.g. `meta-raspberrypi` and `meta-intel`). Each of these layers is a repository unto itself and clicking on a layer reveals information that includes two links from which you can choose to set up a clone of the layer's repository on your local host system. Here is an example that clones the Raspberry Pi BSP layer:

```
$ git clone git://git.yoctoproject.org/meta-raspberrypi
```

In addition to BSP layers, the `meta-yocto-bsp` layer is part of the shipped `poky` repository. The `meta-yocto-bsp` layer maintains several BSPs such as the Beaglebone, EdgeRouter, and generic versions of both 32-bit and 64-bit IA machines.

For information on the BSP development workflow, see the "[Developing a Board Support Package \(BSP\)](#)" section. For more information on how to set up a local copy of source files from a Git repository, see the "[Locating Yocto Project Source Files](#)" section in the Yocto Project Development Tasks Manual.

The layer's base directory (`meta-bsp_root_name`) is the root directory of the BSP Layer. This directory is what you add to the `BBLAYERS` variable in the `conf/bblayers.conf` file found in the [Build Directory](#), which is established after you run the OpenEmbedded build environment setup script (i.e. `oe-init-build-env`). Adding the root directory allows the [OpenEmbedded build system](#) to recognize the BSP layer and from it build an image. Here is an example:

```
BBLAYERS ?= " \
 /usr/local/src/yocto/meta \
 /usr/local/src/yocto/meta-poky \
 /usr/local/src/yocto/meta-yocto-bsp \
```

```
/usr/local/src/yocto/meta-mylayer \
"
```

## Tip

Ordering and `BFILE_PRIORITY` for the layers listed in `BBLAYERS` matter. For example, if multiple layers define a machine configuration, the OpenEmbedded build system uses the last layer searched given similar layer priorities. The build system works from the top-down through the layers listed in `BBLAYERS`.

Some BSPs require or depend on additional layers beyond the BSP's root layer in order to be functional. In this case, you need to specify these layers in the `README` "Dependencies" section of the BSP's root layer. Additionally, if any build instructions exist for the BSP, you must add them to the "Dependencies" section.

Some layers function as a layer to hold other BSP layers. These layers are known as "[container layers](#)". An example of this type of layer is OpenEmbedded's `meta-openembedded` layer. The `meta-openembedded` layer contains many `meta-*` layers.

For more information on layers, see the "[Understanding and Creating Layers](#)" section of the Yocto Project Development Tasks Manual.

## 13.2. Preparing Your Build Host to Work With BSP Layers¶

This section describes how to get your build host ready to work with BSP layers. Once you have the host set up, you can create the layer as described in the "[Creating a new BSP Layer Using the bitbake-layers Script](#)" section.

## Note

For structural information on BSPs, see the [Example Filesystem Layout](#) section.

1. **Set Up the Build Environment:** Be sure you are set up to use BitBake in a shell. See the "[Preparing the Build Host](#)" section in the Yocto Project Development Tasks Manual for information on how to get a build host ready that is either a native Linux machine or a machine that uses CROPS.
2. **Clone the poky Repository:** You need to have a local copy of the Yocto Project [Source Directory](#) (i.e. a local poky repository). See the "[Cloning the poky Repository](#)" and possibly the "[Checking Out by Branch in Poky](#)" or "[Checking Out by Tag in Poky](#)" sections all in the Yocto Project Development Tasks Manual for information on how to clone the poky repository and check out the appropriate branch for your work.
3. **Determine the BSP Layer You Want:** The Yocto Project supports many BSPs, which are maintained in their own layers or in layers designed to contain several BSPs. To get an idea of machine support through BSP layers, you can look at the [index of machines](#) for the release.
4. **Optionally Clone the meta-intel BSP Layer:** If your hardware is based on current Intel CPUs and devices, you can leverage this BSP layer. For details on the `meta-intel` BSP layer, see the layer's [README](#) file.

- a. **Navigate to Your Source Directory:** Typically, you set up the `meta-intel` Git repository inside the [Source Directory](#) (e.g. poky).

```
$ cd /home/you/poky
```

- b. **Clone the Layer:**

```
$ git clone git://git.yoctoproject.org/meta-intel.git
Cloning into 'meta-intel'...
remote: Counting objects: 15585, done.
remote: Compressing objects: 100% (5056/5056), done.
remote: Total 15585 (delta 9123), reused 15329 (delta 8867)
Receiving objects: 100% (15585/15585), 4.51 MiB | 3.19 MiB/s, done.
Resolving deltas: 100% (9123/9123), done.
Checking connectivity... done.
```

- c. **Check Out the Proper Branch:** The branch you check out for `meta-intel` must match the same branch you are using for the Yocto Project release (e.g. zeus):

```
$ cd meta-intel
$ git checkout -b zeus remotes/origin/zeus
Branch zeus set up to track remote branch zeus from origin.
Switched to a new branch 'zeus'
```

## Note

To see the available branch names in a cloned repository, use the `git branch -al` command. See the "[Checking Out By Branch in Poky](#)" section in the Yocto Project Development Tasks Manual for more information.

5. **Optionally Set Up an Alternative BSP Layer:** If your hardware can be more closely leveraged to an existing BSP not within the `meta-intel` BSP layer, you can clone that BSP layer.

The process is identical to the process used for the `meta-intel` layer except for the layer's name. For example, if you determine that your hardware most closely matches the `meta-raspberrypi`, clone that layer:

```
$ git clone git://git.yoctoproject.org/meta-raspberrypi
Cloning into 'meta-raspberrypi'...
remote: Counting objects: 4743, done.
remote: Compressing objects: 100% (2185/2185), done.
remote: Total 4743 (delta 2447), reused 4496 (delta 2258)
Receiving objects: 100% (4743/4743), 1.18 MiB | 0 bytes/s, done.
Resolving deltas: 100% (2447/2447), done.
Checking connectivity... done.
```

6. **Initialize the Build Environment:** While in the root directory of the Source Directory (i.e. poky), run the `oe-init-build-env` environment setup script to define the OpenEmbedded build environment on your build host.

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

Among other things, the script creates the [Build Directory](#), which is `build` in this case and is located in the [Source Directory](#). After the script runs, your current working directory is set to the `build` directory.

### 13.3. Example Filesystem Layout¶

Defining a common BSP directory structure allows end-users to understand and become familiar with that standard. A common format also encourages standardization of software support for hardware.

The proposed form described in this section does have elements that are specific to the OpenEmbedded build system. It is intended that developers can use this structure with other build systems besides the OpenEmbedded build system. It is also intended that it will be simple to extract information and convert it to other formats if required. The OpenEmbedded build system, through its standard [layers mechanism](#), can directly accept the format described as a layer. The BSP layer captures all the hardware-specific details in one place using a standard format, which is useful for any person wishing to use the hardware platform regardless of the build system they are using.

The BSP specification does not include a build system or other tools - the specification is concerned with the hardware-specific components only. At the end-distribution point, you can ship the BSP layer combined with a build system and other tools. Realize that it is important to maintain the distinction that the BSP layer, a build system, and tools are separate components that could be combined in certain end products.

Before looking at the common form for the file structure inside a BSP Layer, you should be aware that some requirements do exist in order for a BSP layer to be considered compliant with the Yocto Project. For that list of requirements, see the "[Released BSP Requirements](#)" section.

Below is the common form for the file structure inside a BSP Layer. While this basic form represents the standard, realize that the actual file structures for specific BSPs could differ.

```
meta-bsp_root_name/
meta-bsp_root_name/bsp_license_file
meta-bsp_root_name/README
meta-bsp_root_name/README.sources
meta-bsp_root_name/binary/bootable_images
meta-bsp_root_name/conf/layer.conf
meta-bsp_root_name/conf/machine/*.conf
meta-bsp_root_name/recipes-bsp/*
meta-bsp_root_name/recipes-core/*
meta-bsp_root_name/recipes-graphics/*
meta-bsp_root_name/recipes-kernel/linux-yocto_kernel_rev.bbappend
```

Below is an example of the Raspberry Pi BSP layer that is available from the [Source Repositories](#):

```
meta-raspberrypi/COPYING.MIT
meta-raspberrypi/README.md
meta-raspberrypi/classes
meta-raspberrypi/classes/sdcard_image-rpi.bbclass
meta-raspberrypi/conf/
meta-raspberrypi/conf/layer.conf
meta-raspberrypi/conf/machine/
meta-raspberrypi/conf/machine/raspberrypi-cm.conf
meta-raspberrypi/conf/machine/raspberrypi-cm3.conf
meta-raspberrypi/conf/machine/raspberrypi.conf
meta-raspberrypi/conf/machine/raspberrypi0-wifi.conf
meta-raspberrypi/conf/machine/raspberrypi0.conf
meta-raspberrypi/conf/machine/raspberrypi2.conf
meta-raspberrypi/conf/machine/raspberrypi3-64.conf
meta-raspberrypi/conf/machine/raspberrypi3.conf
meta-raspberrypi/conf/machine/include
meta-raspberrypi/conf/machine/include/rpi-base.inc
meta-raspberrypi/conf/machine/include/rpi-default-providers.inc
meta-raspberrypi/conf/machine/include/rpi-default-settings.inc
meta-raspberrypi/conf/machine/include/rpi-default-versions.inc
meta-raspberrypi/conf/machine/include/tune-arm1176jzf-s.inc
meta-raspberrypi/docs
meta-raspberrypi/docs/Makefile
meta-raspberrypi/docs/conf.py
meta-raspberrypi/docs/contributing.md
meta-raspberrypi/docs/extra-apps.md
meta-raspberrypi/docs/extra-build-config.md
meta-raspberrypi/docs/index.rst
meta-raspberrypi/docs/layer-contents.md
meta-raspberrypi/docs/readme.md
meta-raspberrypi/files
meta-raspberrypi/files/custom-licenses
meta-raspberrypi/files/custom-licenses/Broadcom
meta-raspberrypi/recipes-bsp
meta-raspberrypi/recipes-bsp/bootfiles
meta-raspberrypi/recipes-bsp/bootfiles/bcm2835-bootfiles.bb
meta-raspberrypi/recipes-bsp/bootfiles/rpi-config_git.bb
meta-raspberrypi/recipes-bsp/common
meta-raspberrypi/recipes-bsp/common/firmware.inc
meta-raspberrypi/recipes-bsp/formfactor
```

```
meta-raspberrypi/recipes-bsp/formfactor/formfactor
meta-raspberrypi/recipes-bsp/formfactor/formfactor/raspberrypi
meta-raspberrypi/recipes-bsp/formfactor/formfactor/raspberrypi/machconfig
meta-raspberrypi/recipes-bsp/formfactor/formfactor_0.0.bbappend
meta-raspberrypi/recipes-bsp/rpi-u-boot-src
meta-raspberrypi/recipes-bsp/rpi-u-boot-src/files
meta-raspberrypi/recipes-bsp/rpi-u-boot-src/files/boot.cmd.in
meta-raspberrypi/recipes-bsp/rpi-u-boot-src/rpi-u-boot-scr.bb
meta-raspberrypi/recipes-bsp/u-boot
meta-raspberrypi/recipes-bsp/u-boot/u-boot
meta-raspberrypi/recipes-bsp/u-boot/u-boot/*.patch
meta-raspberrypi/recipes-bsp/u-boot/u-boot_%.bbappend
meta-raspberrypi/recipes-connectivity
meta-raspberrypi/recipes-connectivity/bluez5
meta-raspberrypi/recipes-connectivity/bluez5/bluez5
meta-raspberrypi/recipes-connectivity/bluez5/bluez5/*.patch
meta-raspberrypi/recipes-connectivity/bluez5/bluez5/BCM43430A1.hcd
meta-raspberrypi/recipes-connectivity/bluez5/bluez5brcm43438.service
meta-raspberrypi/recipes-connectivity/bluez5/bluez5_%.bbappend
meta-raspberrypi/recipes-core
meta-raspberrypi/recipes-core/images
meta-raspberrypi/recipes-core/images/rpi-basic-image.bb
meta-raspberrypi/recipes-core/images/rpi-hwup-image.bb
meta-raspberrypi/recipes-core/images/rpi-test-image.bb
meta-raspberrypi/recipes-core/packagegroups
meta-raspberrypi/recipes-core/packagegroups/packagegroup-rpi-test.bb
meta-raspberrypi/recipes-core/psplash
meta-raspberrypi/recipes-core/psplash/files
meta-raspberrypi/recipes-core/psplash/files/psplash-raspberrypi-img.h
meta-raspberrypi/recipes-core/psplash/psplash_git.bbappend
meta-raspberrypi/recipes-core/udev
meta-raspberrypi/recipes-core/udev/udev-rules-rpi
meta-raspberrypi/recipes-core/udev/udev-rules-rpi/99-com.rules
meta-raspberrypi/recipes-core/udev/udev-rules-rpi.bb
meta-raspberrypi/recipes-devtools
meta-raspberrypi/recipes-devtools/bcm2835
meta-raspberrypi/recipes-devtools/bcm2835/bcm2835_1.52.bb
meta-raspberrypi/recipes-devtools/pi-blast
meta-raspberrypi/recipes-devtools/pi-blast/files
meta-raspberrypi/recipes-devtools/pi-blast/files/*.patch
meta-raspberrypi/recipes-devtools/pi-blast/pi-blast_git.bb
meta-raspberrypi/recipes-devtools/python
meta-raspberrypi/recipes-devtools/python/python-rtmu
meta-raspberrypi/recipes-devtools/python/python-rtmu/*.patch
meta-raspberrypi/recipes-devtools/python/python-rtmu_git.bb
meta-raspberrypi/recipes-devtools/python/python-sense-hat_2.2.0.bb
meta-raspberrypi/recipes-devtools/python/rpi-gpio
meta-raspberrypi/recipes-devtools/python/rpi-gpio/*.patch
meta-raspberrypi/recipes-devtools/python/rpi-gpio_0.6.3.bb
meta-raspberrypi/recipes-devtools/python/rpio
meta-raspberrypi/recipes-devtools/python/rpio/*.patch
meta-raspberrypi/recipes-devtools/python/rpio_0.10.0.bb
meta-raspberrypi/recipes-devtools/wiringPi
meta-raspberrypi/recipes-devtools/wiringPi/files
meta-raspberrypi/recipes-devtools/wiringPi/files/*.patch
meta-raspberrypi/recipes-devtools/wiringPi/wiringpi_git.bb
meta-raspberrypi/recipes-graphics
meta-raspberrypi/recipes-graphics/eglinfo
meta-raspberrypi/recipes-graphics/eglinfo/eglinfo-fb_%.bbappend
meta-raspberrypi/recipes-graphics/eglinfo/eglinfo-x11_%.bbappend
meta-raspberrypi/recipes-graphics/mesa
meta-raspberrypi/recipes-graphics/mesa/mesa-gl_%.bbappend
meta-raspberrypi/recipes-graphics/mesa/mesa_%.bbappend
meta-raspberrypi/recipes-graphics/userland
meta-raspberrypi/recipes-graphics/userland/userland
meta-raspberrypi/recipes-graphics/userland/userland/*.patch
meta-raspberrypi/recipes-graphics/userland/userland_git.bb
meta-raspberrypi/recipes-graphics/vc-graphics
meta-raspberrypi/recipes-graphics/vc-graphics/files
meta-raspberrypi/recipes-graphics/vc-graphics/files/egl.pc
meta-raspberrypi/recipes-graphics/vc-graphics/files/vchiq.sh
meta-raspberrypi/recipes-graphics/vc-graphics/vc-graphics-hardfp.bb
meta-raspberrypi/recipes-graphics/vc-graphics/vc-graphics.bb
meta-raspberrypi/recipes-graphics/vc-graphics/vc-graphics.inc
meta-raspberrypi/recipes-graphics/wayland
meta-raspberrypi/recipes-graphics/wayland/weston_%.bbappend
meta-raspberrypi/recipes-graphics/xorg-xserver
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xf86-config
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xf86-config/rpi
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xf86-config/rpi/xorg.conf
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xf86-config/rpi/xorg.conf.d
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xf86-config/rpi/xorg.conf.d/10-evdev.conf
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xf86-config/rpi/xorg.conf.d/98-pitft.conf
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xf86-config/rpi/xorg.conf.d/99-calibration.conf
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xf86-config_0.1.bbappend
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xorg_%.bbappend
meta-raspberrypi/recipes-kernel
meta-raspberrypi/recipes-kernel/linux-firmware
meta-raspberrypi/recipes-kernel/linux-firmware/files
meta-raspberrypi/recipes-kernel/linux-firmware/files/brcmfmac43430-sdio.bin
meta-raspberrypi/recipes-kernel/linux-firmware/files/brcmfmac43430-sdio.txt
meta-raspberrypi/recipes-kernel/linux-firmware/linux-firmware_%.bbappend
meta-raspberrypi/recipes-kernel/linux
meta-raspberrypi/recipes-kernel/linux/linux-raspberrypi-dev.bb
meta-raspberrypi/recipes-kernel/linux/linux-raspberrypi.inc
meta-raspberrypi/recipes-kernel/linux/linux-raspberrypi_4.14.bb
meta-raspberrypi/recipes-kernel/linux/linux-raspberrypi_4.9.bb
meta-raspberrypi/recipes-multimedia
meta-raspberrypi/recipes-multimedia/gstreamer
meta-raspberrypi/recipes-multimedia/gstreamer/gstreamer1.0-omx
meta-raspberrypi/recipes-multimedia/gstreamer/gstreamer1.0-omx/*.patch
meta-raspberrypi/recipes-multimedia/gstreamer/gstreamer1.0-omx_%.bbappend
meta-raspberrypi/recipes-multimedia/gstreamer/gstreamer1.0-plugins-bad_%.bbappend
```

```
meta-raspberrypi/recipes-multimedia/gstreamer/gstreamer1.0-omx-1.12
meta-raspberrypi/recipes-multimedia/gstreamer/gstreamer1.0-omx-1.12/*.patch
meta-raspberrypi/recipes-multimedia/omxplayer
meta-raspberrypi/recipes-multimedia/omxplayer/omxplayer
meta-raspberrypi/recipes-multimedia/omxplayer/omxplayer/*.patch
meta-raspberrypi/recipes-multimedia/omxplayer/omxplayer_git.bb
meta-raspberrypi/recipes-multimedia/x264
meta-raspberrypi/recipes-multimedia/x264/x264_git.bbappend
meta-raspberrypi/wic
meta-raspberrypi/wic/sdimage-raspberrypi.wks
```

The following sections describe each part of the proposed BSP format.

### 13.3.1. License Files¶

You can find these files in the BSP Layer at:

```
meta-bsp_root_name/bsp_license_file
```

These optional files satisfy licensing requirements for the BSP. The type or types of files here can vary depending on the licensing requirements. For example, in the Raspberry Pi BSP all licensing requirements are handled with the `COPYING.MIT` file.

Licensing files can be MIT, BSD, GPLv\*, and so forth. These files are recommended for the BSP but are optional and totally up to the BSP developer. For information on how to maintain license compliance, see the "[Maintaining Open Source License Compliance During Your Product's Lifecycle](#)" section in the Yocto Project Development Tasks Manual.

### 13.3.2. README File¶

You can find this file in the BSP Layer at:

```
meta-bsp_root_name/README
```

This file provides information on how to boot the live images that are optionally included in the `binary/` directory. The `README` file also provides information needed for building the image.

At a minimum, the `README` file must contain a list of dependencies, such as the names of any other layers on which the BSP depends and the name of the BSP maintainer with his or her contact information.

### 13.3.3. README.sources File¶

You can find this file in the BSP Layer at:

```
meta-bsp_root_name/README.sources
```

This file provides information on where to locate the BSP source files used to build the images (if any) that reside in `meta-bsp_root_name/binary`. Images in the `binary` would be images released with the BSP. The information in the `README.sources` file also helps you find the `Metadata` used to generate the images that ship with the BSP.

## Note

If the BSP's `binary` directory is missing or the directory has no images, an existing `README.sources` file is meaningless and usually does not exist.

### 13.3.4. Pre-built User Binaries¶

You can find these files in the BSP Layer at:

```
meta-bsp_root_name/binary/bootable_images
```

This optional area contains useful pre-built kernels and user-space filesystem images released with the BSP that are appropriate to the target system. This directory typically contains graphical (e.g. Sato) and minimal live images when the BSP tarball has been created and made available in the [Yocto Project](#) website. You can use these kernels and images to get a system running and quickly get started on development tasks.

The exact types of binaries present are highly hardware-dependent. The `README` file should be present in the BSP Layer and it explains how to use the images with the target hardware. Additionally, the `README.sources` file should be present to locate the sources used to build the images and provide information on the `Metadata`.

### 13.3.5. Layer Configuration File¶

You can find this file in the BSP Layer at:

```
meta-bsp_root_name/conf/layer.conf
```

The `conf/layer.conf` file identifies the file structure as a layer, identifies the contents of the layer, and contains information about how the build system should use it. Generally, a standard boilerplate file such as the following works. In the following example, you would replace `bsp` with the actual name of the BSP (i.e. `bsp_root_name` from the example template).

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

We have a recipes directory, add to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
```

```

 ${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "bsp"
BBFILE_PATTERN_bsp = "^${LAYERDIR}/"
BBFILE_PRIORITY_bsp = "6"

LAYERDEPENDS_bsp = "intel"

```

To illustrate the string substitutions, here are the corresponding statements from the Raspberry Pi `conf/layer.conf` file:

```

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

We have a recipes directory containing .bb and .bbappend files, add to BBFILES
BBFILES += "${LAYERDIR}/recipes*/*/*.bb \
 ${LAYERDIR}/recipes*/*/*.bbappend"

BBFILE_COLLECTIONS += "raspberrypi"
BBFILE_PATTERN_raspberrypi := "${LAYERDIR}/"
BBFILE_PRIORITY_raspberrypi = "9"

Additional license directories.
LICENSE_PATH += "${LAYERDIR}/files/custom-licenses"
.
.
.

```

This file simply makes [BitBake](#) aware of the recipes and configuration directories. The file must exist so that the OpenEmbedded build system can recognize the BSP.

### 13.3.6. Hardware Configuration Options¶

You can find these files in the BSP Layer at:

```
meta-bsp_root_name/conf/machine/*.conf
```

The machine files bind together all the information contained elsewhere in the BSP into a format that the build system can understand. Each BSP Layer requires at least one machine file. If the BSP supports multiple machines, multiple machine configuration files can exist. These filenames correspond to the values to which users have set the [MACHINE](#) variable.

These files define things such as the kernel package to use ([PREFERRED\\_PROVIDER](#) of `virtual/kernel`), the hardware drivers to include in different types of images, any special software components that are needed, any bootloader information, and also any special image format requirements.

This configuration file could also include a hardware "tuning" file that is commonly used to define the package architecture and specify optimization flags, which are carefully chosen to give best performance on a given processor.

Tuning files are found in the `meta/conf/machine/include` directory within the [Source Directory](#). For example, many `tune-*` files (e.g. `tune-arm1136jf-s.inc`, `tun-1586-nlp.inc`, and so forth) reside in the `poky/meta/conf/machine/include` directory.

To use an include file, you simply include them in the machine configuration file. For example, the Raspberry Pi BSP `raspberrypi3.conf` contains the following statement:

```
include conf/machine/include/rpi-base.inc
```

### 13.3.7. Miscellaneous BSP-Specific Recipe Files¶

You can find these files in the BSP Layer at:

```
meta-bsp_root_name/recipes-bsp/*
```

This optional directory contains miscellaneous recipe files for the BSP. Most notably would be the formfactor files. For example, in the Raspberry Pi BSP there is the `formfactor_0.0.bbappend` file, which is an append file used to augment the recipe that starts the build. Furthermore, there are machine-specific settings used during the build that are defined by the `machconfig` file further down in the directory. Here is the `machconfig` file for the Raspberry Pi BSP:

```

HAVE_TOUCHSCREEN=0
HAVE_KEYBOARD=1

DISPLAY_CAN_ROTATE=0
DISPLAY_ORIENTATION=0
DISPLAY_DPI=133

```

#### Note

If a BSP does not have a formfactor entry, defaults are established according to the formfactor configuration file that is installed by the main formfactor recipe `meta/recipes-bsp/formfactor/formfactor_0.0.bb`, which is found in the [Source Directory](#).

### 13.3.8. Display Support Files¶

You can find these files in the BSP Layer at:

```
meta-bsp_root_name/recipes-graphics/*
```



This optional directory contains recipes for the BSP if it has special requirements for graphics support. All files that are needed for the BSP to support a display are kept here.

### 13.3.9. Linux Kernel Configuration¶

You can find these files in the BSP Layer at:

```
meta-bsp_root_name/recipes-kernel/linux/linux*.bbappend
meta-bsp_root_name/recipes-kernel/linux/*.bb
```

Append files (`*.bbappend`) modify the main kernel recipe being used to build the image. The `*.bb` files would be a developer-supplied kernel recipe. This area of the BSP hierarchy can contain both these types of files, although in practice, it is likely that you would have one or the other.

For your BSP, you typically want to use an existing Yocto Project kernel recipe found in the [Source Directory](#) at `meta/recipes-kernel/linux`. You can append machine-specific changes to the kernel recipe by using a similarly named append file, which is located in the BSP Layer for your target device (e.g. the `meta-bsp_root_name/recipes-kernel/linux` directory).

Suppose you are using the `linux-yocto_4.4.bb` recipe to build the kernel. In other words, you have selected the kernel in your `bsp_root_name.conf` file by adding `PREFERRED_PROVIDER` and `PREFERRED_VERSION` statements as follows:

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
PREFERRED_VERSION_linux-yocto ?= "4.4%"
```

#### Note

When the preferred provider is assumed by default, the `PREFERRED_PROVIDER` statement does not appear in the `bsp_root_name.conf` file.

You would use the `linux-yocto_4.4.bbappend` file to append specific BSP settings to the kernel, thus configuring the kernel for your particular BSP.

You can find more information on what your append file should contain in the "[Creating the Append File](#)" section in the Yocto Project Linux Kernel Development Manual.

An alternate scenario is when you create your own kernel recipe for the BSP. A good example of this is the Raspberry Pi BSP. If you examine the `recipes-kernel/linux` directory you see the following:

```
linux-raspberrypi-dev.bb
linux-raspberrypi.inc
linux-raspberrypi_4.14.bb
linux-raspberrypi_4.9.bb
```

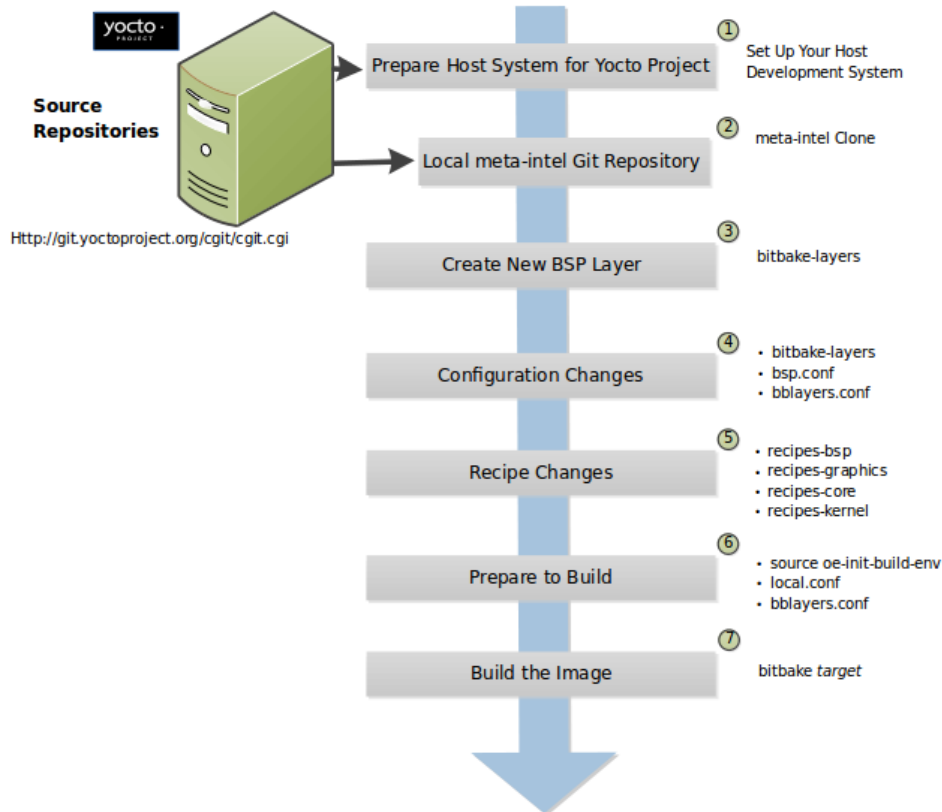
The directory contains three kernel recipes and a common include file.

### 13.4. Developing a Board Support Package (BSP)¶

This section contains the high-level procedure you can follow to create a BSP. Although not required for BSP creation, the `meta-intel` repository, which contains many BSPs supported by the Yocto Project, is part of the example.

For an example that shows how to create a new layer using the tools, see the "[Creating a New BSP Layer Using the `bitbake-layers` Script](#)" section.

The following illustration and list summarize the BSP creation general workflow.



1. **Set up Your Host Development System to Support Development Using the Yocto Project:** See the "[Preparing the Build Host](#)" section in the Yocto Project Development Tasks Manual for options on how to get a system ready to use the Yocto Project.
2. **Establish the meta-intel Repository on Your System:** Having local copies of these supported BSP layers on your system gives you access to layers you might be able to leverage when creating your BSP. For information on how to get these files, see the "[Preparing Your Build Host to Work with BSP Layers](#)" section.
3. **Create Your Own BSP Layer Using the bitbake-layers Script:** Layers are ideal for isolating and storing work for a given piece of hardware. A layer is really just a location or area in which you place the recipes and configurations for your BSP. In fact, a BSP is, in itself, a special type of layer. The simplest way to create a new BSP layer that is compliant with the Yocto Project is to use the `bitbake-layers` script. For information about that script, see the "[Creating a New BSP Layer Using the bitbake-layers Script](#)" section.

Another example that illustrates a layer is an application. Suppose you are creating an application that has library or other dependencies in order for it to compile and run. The layer, in this case, would be where all the recipes that define those dependencies are kept. The key point for a layer is that it is an isolated area that contains all the relevant information for the project that the OpenEmbedded build system knows about. For more information on layers, see the "[The Yocto Project Layer Model](#)" section in the Yocto Project Overview and Concepts Manual. You can also reference the "[Understanding and Creating Layers](#)" section in the Yocto Project Development Tasks Manual. For more information on BSP layers, see the "[BSP Layers](#)" section.

## Notes

- Five hardware reference BSPs exist that are part of the Yocto Project release and are located in the `poky/meta-yocto-bsp` BSP layer:
  - Texas Instruments Beaglebone (`beaglebone-yocto`)
  - Freescale MPC8315E-RDB (`mpc8315e-rdb`)
  - Ubiquiti Networks EdgeRouter Lite (`edgerouter`)
  - Two general IA platforms (`genericx86` and `genericx86-64`)
- Three core Intel BSPs exist as part of the Yocto Project release in the `meta-intel` layer:
  - `intel-core2-32`, which is a BSP optimized for the Core2 family of CPUs as well as all CPUs prior to the Silvermont core.
  - `intel-corei7-64`, which is a BSP optimized for Nehalem and later Core and Xeon CPUs as well as Silvermont and later Atom CPUs, such as the Baytrail SoCs.
  - `intel-quark`, which is a BSP optimized for the Intel Galileo gen1 & gen2 development boards.

When you set up a layer for a new BSP, you should follow a standard layout. This layout is described in the "[Example Filesystem Layout](#)" section. In the standard layout, notice the suggested structure for recipes and configuration information. You can see the standard layout for a BSP by examining any supported BSP found in the `meta-intel` layer inside the Source Directory.

4. **Make Configuration Changes to Your New BSP Layer:** The standard BSP layer structure organizes the files you need to edit in `conf` and several `recipes-*` directories within the BSP layer. Configuration changes identify where your new layer is on the local system and identifies the kernel you are going to use. When you run the `bitbake-layers` script, you are able to interactively configure many things for the BSP (e.g. keyboard, touchscreen, and so forth).

5. **Make Recipe Changes to Your New BSP Layer:** Recipe changes include altering recipes (`*.bb` files), removing recipes you do not use, and adding new recipes or append files (`.bbappend`) that support your hardware.
6. **Prepare for the Build:** Once you have made all the changes to your BSP layer, there remains a few things you need to do for the OpenEmbedded build system in order for it to create your image. You need to get the build environment ready by sourcing an environment setup script (i.e. `oe-init-build-env`) and you need to be sure two key configuration files are configured appropriately: the `conf/local.conf` and the `conf/bblayers.conf` file. You must make the OpenEmbedded build system aware of your new layer. See the "[Enabling Your Layer](#)" section in the Yocto Project Development Tasks Manual for information on how to let the build system know about your new layer.
7. **Build the Image:** The OpenEmbedded build system uses the BitBake tool to build images based on the type of image you want to create. You can find more information about BitBake in the BitBake User Manual.

The build process supports several types of images to satisfy different needs. See the "[Images](#)" chapter in the Yocto Project Reference Manual for information on supported images.

## 13.5. Requirements and Recommendations for Released BSPs¶

Certain requirements exist for a released BSP to be considered compliant with the Yocto Project. Additionally, recommendations also exist. This section describes the requirements and recommendations for released BSPs.

### 13.5.1. Released BSP Requirements¶

Before looking at BSP requirements, you should consider the following:

- The requirements here assume the BSP layer is a well-formed, "legal" layer that can be added to the Yocto Project. For guidelines on creating a layer that meets these base requirements, see the "[BSP Layers](#)" section in this manual and the "[Understanding and Creating Layers](#)" section in the Yocto Project Development Tasks Manual.
- The requirements in this section apply regardless of how you package a BSP. You should consult the packaging and distribution guidelines for your specific release process. For an example of packaging and distribution requirements, see the "[Third Party BSP Release Process](#)" wiki page.
- The requirements for the BSP as it is made available to a developer are completely independent of the released form of the BSP. For example, the BSP Metadata can be contained within a Git repository and could have a directory structure completely different from what appears in the officially released BSP layer.
- It is not required that specific packages or package modifications exist in the BSP layer, beyond the requirements for general compliance with the Yocto Project. For example, no requirement exists dictating that a specific kernel or kernel version be used in a given BSP.

Following are the requirements for a released BSP that conform to the Yocto Project:

- **Layer Name:** The BSP must have a layer name that follows the Yocto Project standards. For information on BSP layer names, see the "[BSP Layers](#)" section.
- **File System Layout:** When possible, use the same directory names in your BSP layer as listed in the `recipes.txt` file, which is found in `poky/meta` directory of the [Source Directory](#) or in the OpenEmbedded-Core Layer (`openembedded-core`) at <http://git.openembedded.org/openembedded-core/tree/meta>.

You should place recipes (`*.bb` files) and recipe modifications (`*.bbappend` files) into `recipes-*` subdirectories by functional area as outlined in `recipes.txt`. If you cannot find a category in `recipes.txt` to fit a particular recipe, you can make up your own `recipes-*` subdirectory.

Within any particular `recipes-*` category, the layout should match what is found in the OpenEmbedded-Core Git repository (`openembedded-core`) or the Source Directory (`poky`). In other words, make sure you place related files in appropriately related `recipes-*` subdirectories specific to the recipe's function, or within a subdirectory containing a set of closely-related recipes. The recipes themselves should follow the general guidelines for recipes used in the Yocto Project found in the "[OpenEmbedded Style Guide](#)".

- **License File:** You must include a license file in the `meta-bsp_root_name` directory. This license covers the BSP Metadata as a whole. You must specify which license to use since no default license exists when one not specified. See the [COPYING.MIT](#) file for the Raspberry Pi BSP in the `meta-raspberrypi` BSP layer as an example.
- **README File:** You must include a README file in the `meta-bsp_root_name` directory. See the [README.md](#) file for the Raspberry Pi BSP in the `meta-raspberrypi` BSP layer as an example.

At a minimum, the README file should contain the following:

- A brief description about the hardware the BSP targets.
- A list of all the dependencies on which a BSP layer depends. These dependencies are typically a list of required layers needed to build the BSP. However, the dependencies should also contain information regarding any other dependencies the BSP might have.
- Any required special licensing information. For example, this information includes information on special variables needed to satisfy a EULA, or instructions on information needed to build or distribute binaries built from the BSP Metadata.
- The name and contact information for the BSP layer maintainer. This is the person to whom patches and questions should be sent. For information on how to find the right person, see the "[Submitting a Change to the Yocto Project](#)" section in the Yocto Project Development Tasks Manual.
- Instructions on how to build the BSP using the BSP layer.
- Instructions on how to boot the BSP build from the BSP layer.
- Instructions on how to boot the binary images contained in the `binary` directory, if present.
- Information on any known bugs or issues that users should know about when either building or booting the BSP binaries.

- **README.sources File:** If your BSP contains binary images in the `binary` directory, you must include a `README.sources` file in the `meta-bsp_root_name` directory. This file specifies exactly where you can find the sources used to generate the binary images.
- **Layer Configuration File:** You must include a `conf/layer.conf` file in the `meta-bsp_root_name` directory. This file identifies the `meta-bsp_root_name` BSP layer as a layer to the build system.
- **Machine Configuration File:** You must include one or more `conf/machine/bsp_root_name.conf` files in the `meta-bsp_root_name` directory. These configuration files define machine targets that can be built using the BSP layer. Multiple machine configuration files define variations of machine configurations that the BSP supports. If a BSP supports multiple machine variations, you need to adequately describe each variation in the BSP README file. Do not use multiple machine configuration files to describe disparate hardware. If you do have very different targets, you should create separate BSP layers for each target.

## Note

It is completely possible for a developer to structure the working repository as a conglomeration of unrelated BSP files, and to possibly generate BSPs targeted for release from that directory using scripts or some other mechanism (e.g. `meta-yocto-bsp` layer). Such considerations are outside the scope of this document.

### 13.5.2. Released BSP Recommendations¶

Following are recommendations for released BSPs that conform to the Yocto Project:

- **Bootable Images:** Released BSPs can contain one or more bootable images. Including bootable images allows users to easily try out the BSP using their own hardware.

In some cases, it might not be convenient to include a bootable image. If so, you might want to make two versions of the BSP available: one that contains binary images, and one that does not. The version that does not contain bootable images avoids unnecessary download times for users not interested in the images.

If you need to distribute a BSP and include bootable images or build kernel and filesystems meant to allow users to boot the BSP for evaluation purposes, you should put the images and artifacts within a `binary/` subdirectory located in the `meta-bsp_root_name` directory.

## Note

If you do include a bootable image as part of the BSP and the image was built by software covered by the GPL or other open source licenses, it is your responsibility to understand and meet all licensing requirements, which could include distribution of source files.

- **Use a Yocto Linux Kernel:** Kernel recipes in the BSP should be based on a Yocto Linux kernel. Basing your recipes on these kernels reduces the costs for maintaining the BSP and increases its scalability. See the `Yocto Linux Kernel` category in the [Source Repositories](#) for these kernels.

### 13.6. Customizing a Recipe for a BSP¶

If you plan on customizing a recipe for a particular BSP, you need to do the following:

- Create a `*.bbappend` file for the modified recipe. For information on using append files, see the ["Using .bbappend Files in Your Layer"](#) section in the Yocto Project Development Tasks Manual.
- Ensure your directory structure in the BSP layer that supports your machine is such that the OpenEmbedded build system can find it. See the example later in this section for more information.
- Put the append file in a directory whose name matches the machine's name and is located in an appropriate sub-directory inside the BSP layer (i.e. `recipes-bsp`, `recipes-graphics`, `recipes-core`, and so forth).
- Place the BSP-specific files in the proper directory inside the BSP layer. How expansive the layer is affects where you must place these files. For example, if your layer supports several different machine types, you need to be sure your layer's directory structure includes hierarchy that separates the files according to machine. If your layer does not support multiple machines, the layer would not have that additional hierarchy and the files would obviously not be able to reside in a machine-specific directory.

Following is a specific example to help you better understand the process. This example customizes a recipe by adding a BSP-specific configuration file named `interfaces` to the `init-ifupdown_1.0.bb` recipe for machine "xyz" where the BSP layer also supports several other machines:

1. Edit the `init-ifupdown_1.0.bbappend` file so that it contains the following:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
```

The append file needs to be in the `meta-xyz/recipes-core/init-ifupdown` directory.

2. Create and place the new `interfaces` configuration file in the BSP's layer here:

```
meta-xyz/recipes-core/init-ifupdown/files/xyz-machine-one/interfaces
```

## Note

If the `meta-xyz` layer did not support multiple machines, you would place the `interfaces` configuration file in the layer here:

The `FILESEXTRAPATHS` variable in the append files extends the search path the build system uses to find files during the build. Consequently, for this example you need to have the `files` directory in the same location as your append file.

### 13.7. BSP Licensing Considerations¶

In some cases, a BSP contains separately licensed Intellectual Property (IP) for a component or components. For these cases, you are required to accept the terms of a commercial or other type of license that requires some kind of explicit End User License Agreement (EULA). Once you accept the license, the OpenEmbedded build system can then build and include the corresponding component in the final BSP image. If the BSP is available as a pre-built image, you can download the image after agreeing to the license or EULA.

You could find that some separately licensed components that are essential for normal operation of the system might not have an unencumbered (or free) substitute. Without these essential components, the system would be non-functional. Then again, you might find that other licensed components that are simply 'good-to-have' or purely elective do have an unencumbered, free replacement component that you can use rather than agreeing to the separately licensed component. Even for components essential to the system, you might find an unencumbered component that is not identical but will work as a less-capable version of the licensed version in the BSP recipe.

For cases where you can substitute a free component and still maintain the system's functionality, the "DOWNLOADS" selection from the "SOFTWARE" tab on the [Yocto Project website](#) makes available de-featured BSPs that are completely free of any IP encumbrances. For these cases, you can use the substitution directly and without any further licensing requirements. If present, these fully de-featured BSPs are named appropriately different as compared to the names of their respective encumbered BSPs. If available, these substitutions are your simplest and most preferred options. Obviously, use of these substitutions assumes the resulting functionality meets system requirements.

#### Note

If however, a non-encumbered version is unavailable or it provides unsuitable functionality or quality, you can use an encumbered version.

A couple different methods exist within the OpenEmbedded build system to satisfy the licensing requirements for an encumbered BSP. The following list describes them in order of preference:

1. **Use the `LICENSE_FLAGS` Variable to Define the Recipes that Have Commercial or Other Types of Specially-Licensed Packages:** For each of those recipes, you can specify a matching license string in a `local.conf` variable named `LICENSE_FLAGS_WHITELIST`. Specifying the matching license string signifies that you agree to the license. Thus, the build system can build the corresponding recipe and include the component in the image. See the "[Enabling Commercially Licensed Recipes](#)" section in the Yocto Project Development Tasks Manual for details on how to use these variables.

If you build as you normally would, without specifying any recipes in the `LICENSE_FLAGS_WHITELIST`, the build stops and provides you with the list of recipes that you have tried to include in the image that need entries in the `LICENSE_FLAGS_WHITELIST`. Once you enter the appropriate license flags into the whitelist, restart the build to continue where it left off. During the build, the prompt will not appear again since you have satisfied the requirement.

Once the appropriate license flags are on the white list in the `LICENSE_FLAGS_WHITELIST` variable, you can build the encumbered image with no change at all to the normal build process.

2. **Get a Pre-Built Version of the BSP:** You can get this type of BSP by selecting the "DOWNLOADS" item from the "SOFTWARE" tab on the [Yocto Project website](#). You can download BSP tarballs that contain proprietary components after agreeing to the licensing requirements of each of the individually encumbered packages as part of the download process. Obtaining the BSP this way allows you to access an encumbered image immediately after agreeing to the click-through license agreements presented by the website. If you want to build the image yourself using the recipes contained within the BSP tarball, you will still need to create an appropriate `LICENSE_FLAGS_WHITELIST` to match the encumbered recipes in the BSP.

#### Note

Pre-compiled images are bundled with a time-limited kernel that runs for a predetermined amount of time (10 days) before it forces the system to reboot. This limitation is meant to discourage direct redistribution of the image. You must eventually rebuild the image if you want to remove this restriction.

### 13.8. Creating a new BSP Layer Using the `bitbake-layers` Script¶

The `bitbake-layers create-layer` script automates creating a BSP layer. What makes a layer a "BSP layer", is the presence of a machine configuration file. Additionally, a BSP layer usually has a kernel recipe or an append file that leverages off an existing kernel recipe. The primary requirement, however, is the machine configuration.

Use these steps to create a BSP layer:

- **Create a General Layer:** Use the `bitbake-layers` script with the `create-layer` subcommand to create a new general layer. For instructions on how to create a general layer using the `bitbake-layers` script, see the "[Creating a General Layer Using the `bitbake-layers` Script](#)" section in the Yocto Project Development Tasks Manual.
- **Create a Layer Configuration File:** Every layer needs a layer configuration file. This configuration file establishes locations for the layer's recipes, priorities for the layer, and so forth. You can find examples of `layer.conf` files in the Yocto Project [Source Repositories](#). To get examples of what you need in your configuration file, locate a layer (e.g. "meta-ti") and examine the <http://git.yoctoproject.org/cgit/cgit.cgi/meta-ti/tree/conf/layer.conf> file.

- **Create a Machine Configuration File:** Create a `conf/machine/bsp_root_name.conf` file. See [meta-yocto-bsp/conf/machine](#) for sample `bsp_root_name.conf` files. Other samples such as [meta-ti](#) and [meta-freescale](#) exist from other vendors that have more specific machine and tuning examples.
- **Create a Kernel Recipe:** Create a kernel recipe in `recipes-kernel/linux` by either using a kernel append file or a new custom kernel recipe file (e.g. `yocto-linux_4.12.bb`). The BSP layers mentioned in the previous step also contain different kernel examples. See the "Modifying an Existing Recipe" section in the Yocto Project Linux Kernel Development Manual for information on how to create a custom kernel.

The remainder of this section provides a description of the Yocto Project reference BSP for Beaglebone, which resides in the [meta-yocto-bsp](#) layer.

### 13.8.1. BSP Layer Configuration Example¶

The layer's `conf` directory contains the `layer.conf` configuration file. In this example, the `conf/layer.conf` is the following:

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

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

BBFILE_COLLECTIONS += "yoctobsp"
BBFILE_PATTERN_yoctobsp = "^${LAYERDIR}/"
BBFILE_PRIORITY_yoctobsp = "5"
LAYERVERSION_yoctobsp = "4"
LAYERSERIES_COMPAT_yoctobsp = "zeus"
```

The variables used in this file configure the layer. A good way to learn about layer configuration files is to examine various files for BSP from the [Source Repositories](#).

For a detailed description of this particular layer configuration file, see "step 3" in the discussion that describes how to create layers in the Yocto Project Development Tasks Manual.

### 13.8.2. BSP Machine Configuration Example¶

As mentioned earlier in this section, the existence of a machine configuration file is what makes a layer a BSP layer as compared to a general or kernel layer.

Machine configuration files exist in the `bsp_layer/conf/machine/` directory of the layer:

```
bsp_layer/conf/machine/machine.conf
```

For example, the machine configuration file for the [BeagleBone and BeagleBone Black development boards](#) is located in the layer `poky/meta-yocto-bsp/conf/machine` and is named `beaglebone-yocto.conf`:

```
#@TYPE: Machine
#@NAME: Beaglebone-yocto machine
#@DESCRIPTION: Reference machine configuration for http://beagleboard.org/bone and http://beagleboard.org/black boards

PREFERRED_PROVIDER_virtual/xserver ?= "xserver-xorg"
XSERVER ?= "xserver-xorg \
 xf86-video-modesetting \
 "

MACHINE_EXTRA_RRECOMMENDS = "kernel-modules kernel-devicetree"

EXTRA_IMAGEDEPENDS += "u-boot"

DEFAULTTUNE ?= "cortexa8hf-neon"
include conf/machine/include/tune-cortexa8.inc

IMAGE_FSTYPES += "tar.bz2 jffs2 wic wic.bmap"
EXTRA_IMAGECMD_jffs2 = "-lnp"
WKS_FILE ?= "beaglebone-yocto.wks"
IMAGE_INSTALL_append = " kernel-devicetree kernel-image-zimage"
do_image_wic[depends] += "mtools-native:do_populate_sysroot dosfstools-native:do_populate_sysroot"

SERIAL_CONSOLES ?= "115200;ttyS0 115200;ttyO0"
SERIAL_CONSOLES_CHECK = "${SERIAL_CONSOLES}"

PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
PREFERRED_VERSION_linux-yocto ?= "5.0%"

KERNEL_IMAGETYPE = "zImage"
KERNEL_DEVICETREE = "am335x-bone.dtb am335x-boneblack.dtb am335x-bonegreen.dtb"
KERNEL_EXTRA_ARGS += "LOADADDR=${UBOOT_ENTRYPOINT}"

SPL_BINARY = "MLO"
UBOOT_SUFFIX = "img"
UBOOT_MACHINE = "am335x_ewm_defconfig"
UBOOT_ENTRYPOINT = "0x80008000"
UBOOT_LOADADDRESS = "0x80008000"

MACHINE_FEATURES = "usb gadget usbhost vfat alsa"

IMAGE_BOOT_FILES ?= "u-boot.${UBOOT_SUFFIX} MLO zImage am335x-bone.dtb am335x-boneblack.dtb am335x-bonegreen.dtb"
```

The variables used to configure the machine define machine-specific properties. For example, machine-dependent packages, machine tunings, the type of kernel to build, and U-Boot configurations.

The following list provides some explanation for the statements found in the example reference machine configuration file for the BeagleBone development boards. Realize that much more can be defined as part of a machine's configuration file. In general, you can learn about related variables that this example does not have by locating the variables in the "[Yocto Project Variables Glossary](#)" in the Yocto Project Reference Manual.

- `PREFERRED_PROVIDER_virtual/xserver`: The recipe that provides "virtual/xserver" when more than one provider is found. In this case, the recipe that provides "virtual/xserver" is "xserver-xorg", which exists in `poky/meta/recipes-graphics/xorg-xserver`.
- `XSERVER`: The packages that should be installed to provide an X server and drivers for the machine. In this example, the "xserver-xorg" and "xf86-video-modesetting" are installed.
- `MACHINE_EXTRA_RECOMMENDS`: A list of machine-dependent packages not essential for booting the image. Thus, the build does not fail if the packages do not exist. However, the packages are required for a fully-featured image.

## Tip

Many `MACHINE*` variables exist that help you configure a particular piece of hardware.

- `EXTRA_IMAGEDEPENDS`: Recipes to build that do not provide packages for installing into the root filesystem but building the image depends on the recipes. Sometimes a recipe is required to build the final image but is not needed in the root filesystem. In this case, the U-Boot recipe must be built for the image.
- `DEFAULTTUNE`: Machines use tunings to optimize machine, CPU, and application performance. These features, which are collectively known as "tuning features", exist in the `OpenEmbedded-Core (OE-Core)` layer (e.g. `poky/meta/conf/machine/include`). In this example, the default tuning file is "cortexa8hf-neon".

## Note

The `include` statement that pulls in the `conf/machine/include/tune-cortexa8.inc` file provides many tuning possibilities.

- `IMAGE_FSTYPES`: The formats the OpenEmbedded build system uses during the build when creating the root filesystem. In this example, four types of images are supported.
- `EXTRA_IMAGECMD`: Specifies additional options for image creation commands. In this example, the "-lmp" option is used when creating the `JFFS2` image.
- `WKS_FILE`: The location of the `Wic` kickstart file used by the OpenEmbedded build system to create a partitioned image (image.wic).
- `IMAGE_INSTALL`: Specifies packages to install into an image through the `image` class. Recipes use the `IMAGE_INSTALL` variable.
- `do_image_wic[depends]`: A task that is constructed during the build. In this example, the task depends on specific tools in order to create the sysroot when building a `Wic` image.
- `SERIAL_CONSOLES`: Defines a serial console (TTY) to enable using `getty`. In this case, the baud rate is "115200" and the device name is "ttyO0".
- `PREFERRED_PROVIDER_virtual/kernel`: Specifies the recipe that provides "virtual/kernel" when more than one provider is found. In this case, the recipe that provides "virtual/kernel" is "linux-yocto", which exists in the layer's `recipes-kernel/linux` directory.
- `PREFERRED_VERSION_linux-yocto`: Defines the version of the recipe used to build the kernel, which is "5.0" in this case.
- `KERNEL_IMAGETYPE`: The type of kernel to build for the device. In this case, the OpenEmbedded build system creates a "zImage" image type.
- `KERNEL_DEVICETREE`: The names of the generated Linux kernel device trees (i.e. the `*.dts` files). All the device trees for the various BeagleBone devices are included.
- `KERNEL_EXTRA_ARGS`: Additional `make` command-line arguments the OpenEmbedded build system passes on when compiling the kernel. In this example, "LOADADDR=\${UBOOT\_ENTRYPOINT}" is passed as a command-line argument.
- `SPL_BINARY`: Defines the Secondary Program Loader (SPL) binary type. In this case, the SPL binary is set to "MLO", which stands for Multimedia card LOader.

The BeagleBone development board requires an SPL to boot and that SPL file type must be MLO. Consequently, the machine configuration needs to define `SPL_BINARY` as "MLO".

## Note

For more information on how the SPL variables are used, see the `u-boot.inc` include file.

- `UBOOT*`: Defines various U-Boot configurations needed to build a U-Boot image. In this example, a U-Boot image is required to boot the BeagleBone device. See the following variables for more information:
  - `UBOOT_SUFFIX`: Points to the generated U-Boot extension.
  - `UBOOT_MACHINE`: Specifies the value passed on the make command line when building a U-Boot image.



- `UBOOT_ENTRYPOINT`: Specifies the entry point for the U-Boot image.
- `UBOOT_LOADADDRESS`: Specifies the load address for the U-Boot image.
- `MACHINE_FEATURES`: Specifies the list of hardware features the BeagleBone device is capable of supporting. In this case, the device supports "usb gadget usb host vfat alsa".
- `IMAGE_BOOT_FILES`: Files installed into the device's boot partition when preparing the image using the Wic tool with the `bootimg-partition` source plugin.

### 13.8.3. BSP Kernel Recipe Example¶

The kernel recipe used to build the kernel image for the BeagleBone device was established in the machine configuration:

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
PREFERRED_VERSION_linux-yocto ?= "5.0%"
```

The `meta-yocto-bsp/recipes-kernel/linux` directory in the layer contains metadata used to build the kernel. In this case, a kernel append file (i.e. `linux-yocto_5.0.bbappend`) is used to override an established kernel recipe (i.e. `linux-yocto_5.0.bb`), which is located in <http://git.yoctoproject.org/cgit/cgit.cgi/poky/tree/meta/recipes-kernel/linux>.

Following is the contents of the append file:

```
KBRANCH_genericx86 = "v5.0/standard/base"
KBRANCH_genericx86-64 = "v5.0/standard/base"
KBRANCH_edgerouter = "v5.0/standard/edgerouter"
KBRANCH_beaglebone-yocto = "v5.0/standard/beaglebone"
KBRANCH_mpc8315e-rdb = "v5.0/standard/fsl-mpc8315e-rdb"

KMACHINE_genericx86 ?= "common-pc"
KMACHINE_genericx86-64 ?= "common-pc-64"
KMACHINE_beaglebone-yocto ?= "beaglebone"

SRCREV_machine_genericx86 ?= "3df4aae6074e94e794e27fe7f17451d9353cdf3d"
SRCREV_machine_genericx86-64 ?= "3df4aae6074e94e794e27fe7f17451d9353cdf3d"
SRCREV_machine_edgerouter ?= "3df4aae6074e94e794e27fe7f17451d9353cdf3d"
SRCREV_machine_beaglebone-yocto ?= "3df4aae6074e94e794e27fe7f17451d9353cdf3d"
SRCREV_machine_mpc8315e-rdb ?= "8b62af7f252af10588276802c4c6d7c502e875be"

COMPATIBLE_MACHINE_genericx86 = "genericx86"
COMPATIBLE_MACHINE_genericx86-64 = "genericx86-64"
COMPATIBLE_MACHINE_edgerouter = "edgerouter"
COMPATIBLE_MACHINE_beaglebone-yocto = "beaglebone-yocto"
COMPATIBLE_MACHINE_mpc8315e-rdb = "mpc8315e-rdb"

LINUX_VERSION_genericx86 = "5.0.3"
LINUX_VERSION_genericx86-64 = "5.0.3"
LINUX_VERSION_edgerouter = "5.0.3"
LINUX_VERSION_beaglebone-yocto = "5.0.3"
LINUX_VERSION_mpc8315e-rdb = "5.0.3"
```

This particular append file works for all the machines that are part of the `meta-yocto-bsp` layer. The relevant statements are appended with the "beaglebone-yocto" string. The OpenEmbedded build system uses these statements to override similar statements in the kernel recipe:

- `KBRANCH`: Identifies the kernel branch that is validated, patched, and configured during the build.
- `KMACHINE`: Identifies the machine name as known by the kernel, which is sometimes a different name than what is known by the OpenEmbedded build system.
- `SRCREV`: Identifies the revision of the source code used to build the image.
- `COMPATIBLE_MACHINE`: A regular expression that resolves to one or more target machines with which the recipe is compatible.
- `LINUX_VERSION`: The Linux version from kernel.org used by the OpenEmbedded build system to build the kernel image.



# Yocto Project Linux Kernel Development Manual

## Chapter 14. Introduction¶

### 14.1. Overview

### 14.2. Kernel Modification Workflow

### 14.1. Overview¶

Regardless of how you intend to make use of the Yocto Project, chances are you will work with the Linux kernel. This manual describes how to set up your build host to support kernel development, introduces the kernel development process, provides background information on the Yocto Linux kernel [Metadata](#), describes common tasks you can perform using the kernel tools, shows you how to use



the kernel Metadata needed to work with the kernel inside the Yocto Project, and provides insight into how the Yocto Project team develops and maintains Yocto Linux kernel Git repositories and Metadata.

Each Yocto Project release has a set of Yocto Linux kernel recipes, whose Git repositories you can view in the Yocto [Source Repositories](#) under the "Yocto Linux Kernel" heading. New recipes for the release track the latest Linux kernel upstream developments from <http://www.kernel.org> and introduce newly-supported platforms. Previous recipes in the release are refreshed and supported for at least one additional Yocto Project release. As they align, these previous releases are updated to include the latest from the Long Term Support Initiative (LTSI) project. You can learn more about Yocto Linux kernels and LTSI in the "[Yocto Project Kernel Development and Maintenance](#)" section.

Also included is a Yocto Linux kernel development recipe (`linux-yocto-dev.bb`) should you want to work with the very latest in upstream Yocto Linux kernel development and kernel Metadata development.

## Note

For more on Yocto Linux kernels, see the "[Yocto Project Kernel Development and Maintenance](#)" section.

The Yocto Project also provides a powerful set of kernel tools for managing Yocto Linux kernel sources and configuration data. You can use these tools to make a single configuration change, apply multiple patches, or work with your own kernel sources.

In particular, the kernel tools allow you to generate configuration fragments that specify only what you must, and nothing more. Configuration fragments only need to contain the highest level visible `CONFIG` options as presented by the Yocto Linux kernel `menuconfig` system. Contrast this against a complete Yocto Linux kernel `.config` file, which includes all the automatically selected `CONFIG` options. This efficiency reduces your maintenance effort and allows you to further separate your configuration in ways that make sense for your project. A common split separates policy and hardware. For example, all your kernels might support the `proc` and `sys` filesystems, but only specific boards require sound, USB, or specific drivers. Specifying these configurations individually allows you to aggregate them together as needed, but maintains them in only one place. Similar logic applies to separating source changes.

If you do not maintain your own kernel sources and need to make only minimal changes to the sources, the released recipes provide a vetted base upon which to layer your changes. Doing so allows you to benefit from the continual kernel integration and testing performed during development of the Yocto Project.

If, instead, you have a very specific Linux kernel source tree and are unable to align with one of the official Yocto Linux kernel recipes, an alternative exists by which you can use the Yocto Project Linux kernel tools with your own kernel sources.

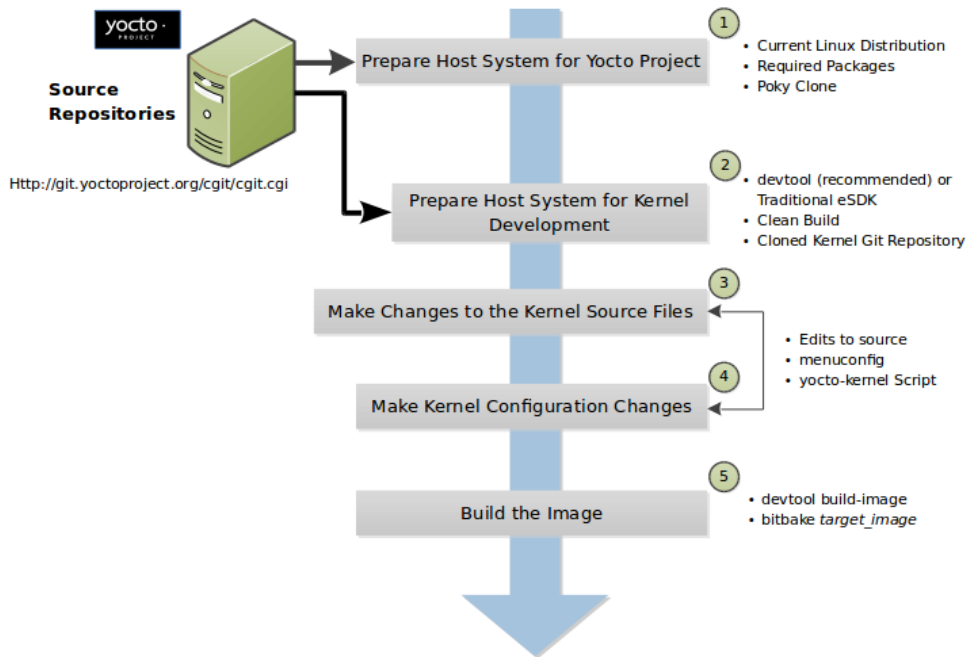
The remainder of this manual provides instructions for completing specific Linux kernel development tasks. These instructions assume you are comfortable working with [BitBake](#) recipes and basic open-source development tools. Understanding these concepts will facilitate the process of working with the kernel recipes. If you find you need some additional background, please be sure to review and understand the following documentation:

- Yocto Project Quick Build document.
- Yocto project Overview and Concepts Manual.
- [devtool workflow](#) as described in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.
- The "[Understanding and Creating Layers](#)" section in the Yocto Project Development Tasks Manual.
- The "[Kernel Modification Workflow](#)" section.

## 14.2. Kernel Modification Workflow¶

Kernel modification involves changing the Yocto Project kernel, which could involve changing configuration options as well as adding new kernel recipes. Configuration changes can be added in the form of configuration fragments, while recipe modification comes through the kernel's `recipes-kernel` area in a kernel layer you create.

This section presents a high-level overview of the Yocto Project kernel modification workflow. The illustration and accompanying list provide general information and references for further information.



- 1. Set up Your Host Development System to Support Development Using the Yocto Project:** See the "[Setting Up the Development Host to Use the Yocto Project](#)" section in the Yocto Project Development Tasks Manual for options on how to get a build host ready to use the Yocto Project.
- 2. Set Up Your Host Development System for Kernel Development:** It is recommended that you use `devtool` and an extensible SDK for kernel development. Alternatively, you can use traditional kernel development methods with the Yocto Project. Either way, there are steps you need to take to get the development environment ready.  
  
Using `devtool` and the eSDK requires that you have a clean build of the image and that you are set up with the appropriate eSDK. For more information, see the "[Getting Ready to Develop Using devtool](#)" section.  
  
Using traditional kernel development requires that you have the kernel source available in an isolated local Git repository. For more information, see the "[Getting Ready for Traditional Kernel Development](#)" section.
- 3. Make Changes to the Kernel Source Code if applicable:** Modifying the kernel does not always mean directly changing source files. However, if you have to do this, you make the changes to the files in the eSDK's Build Directory if you are using `devtool`. For more information, see the "[Using devtool to Patch the Kernel](#)" section.  
  
If you are using traditional kernel development, you edit the source files in the kernel's local Git repository. For more information, see the "[Using Traditional Kernel Development to Patch the Kernel](#)" section.
- 4. Make Kernel Configuration Changes if Applicable:** If your situation calls for changing the kernel's configuration, you can use `menuconfig`, which allows you to interactively develop and test the configuration changes you are making to the kernel. Saving changes you make with `menuconfig` updates the kernel's `.config` file.

## Warning

Try to resist the temptation to directly edit an existing `.config` file, which is found in the Build Directory among the source code used for the build. Doing so, can produce unexpected results when the OpenEmbedded build system regenerates the configuration file.

Once you are satisfied with the configuration changes made using `menuconfig` and you have saved them, you can directly compare the resulting `.config` file against an existing original and gather those changes into a [configuration fragment file](#) to be referenced from within the kernel's `.bbappend` file.

Additionally, if you are working in a BSP layer and need to modify the BSP's kernel's configuration, you can use `menuconfig`.

- 5. Rebuild the Kernel Image With Your Changes:** Rebuilding the kernel image applies your changes. Depending on your target hardware, you can verify your changes on actual hardware or perhaps QEMU.

The remainder of this developer's guide covers common tasks typically used during kernel development, advanced Metadata usage, and Yocto Linux kernel maintenance concepts.

## Chapter 15. Common Tasks¶

- [15.1. Preparing the Build Host to Work on the Kernel](#)
  - [15.1.1. Getting Ready to Develop Using devtool](#)
  - [15.1.2. Getting Ready for Traditional Kernel Development](#)
- [15.2. Creating and Preparing a Layer](#)
- [15.3. Modifying an Existing Recipe](#)
  - [15.3.1. Creating the Append File](#)
  - [15.3.2. Applying Patches](#)
  - [15.3.3. Changing the Configuration](#)

- [15.3.4. Using an "In-Tree" `defconfig` File](#)
- [15.4. Using `devtool` to Patch the Kernel](#)
- [15.5. Using Traditional Kernel Development to Patch the Kernel](#)
- [15.6. Configuring the Kernel](#)
  - [15.6.1. Using `menuconfig`](#)
  - [15.6.2. Creating a `defconfig` File](#)
  - [15.6.3. Creating Configuration Fragments](#)
  - [15.6.4. Validating Configuration](#)
  - [15.6.5. Fine-Tuning the Kernel Configuration File](#)
- [15.7. Expanding Variables](#)
- [15.8. Working with a "Dirty" Kernel Version String](#)
- [15.9. Working With Your Own Sources](#)
- [15.10. Working with Out-of-Tree Modules](#)
  - [15.10.1. Building Out-of-Tree Modules on the Target](#)
  - [15.10.2. Incorporating Out-of-Tree Modules](#)
- [15.11. Inspecting Changes and Commits](#)
  - [15.11.1. What Changed in a Kernel?](#)
  - [15.11.2. Showing a Particular Feature or Branch Change](#)
- [15.12. Adding Recipe-Space Kernel Features](#)

This chapter presents several common tasks you perform when you work with the Yocto Project Linux kernel. These tasks include preparing your host development system for kernel development, preparing a layer, modifying an existing recipe, patching the kernel, configuring the kernel, iterative development, working with your own sources, and incorporating out-of-tree modules.

## Note

The examples presented in this chapter work with the Yocto Project 2.4 Release and forward.

## 15.1. Preparing the Build Host to Work on the Kernel¶

Before you can do any kernel development, you need to be sure your build host is set up to use the Yocto Project. For information on how to get set up, see the ["Preparing the Build Host"](#) section in the Yocto Project Development Tasks Manual. Part of preparing the system is creating a local Git repository of the [Source Directory](#) (`poky`) on your system. Follow the steps in the ["Cloning the `poky` Repository"](#) section in the Yocto Project Development Tasks Manual to set up your Source Directory.

## Note

Be sure you check out the appropriate development branch or you create your local branch by checking out a specific tag to get the desired version of Yocto Project. See the ["Checking Out by Branch in Poky"](#) and ["Checking Out by Tag in Poky"](#) sections in the Yocto Project Development Tasks Manual for more information.

Kernel development is best accomplished using [devtool](#) and not through traditional kernel workflow methods. The remainder of this section provides information for both scenarios.

### 15.1.1. Getting Ready to Develop Using `devtool`¶

Follow these steps to prepare to update the kernel image using `devtool`. Completing this procedure leaves you with a clean kernel image and ready to make modifications as described in the ["Using `devtool` to Patch the Kernel"](#) section:

1. **Initialize the BitBake Environment:** Before building an extensible SDK, you need to initialize the BitBake build environment by sourcing the build environment script (i.e. [oe-init-build-env](#)):

```
$ cd ~/poky
$ source oe-init-build-env
```

## Note

The previous commands assume the [Source Repositories](#) (i.e. `poky`) have been cloned using Git and the local repository is named "poky".

2. **Prepare Your `local.conf` File:** By default, the [MACHINE](#) variable is set to "qemux86", which is fine if you are building for the QEMU emulator in 32-bit mode. However, if you are not, you need to set the `MACHINE` variable appropriately in your `conf/local.conf` file found in the [Build Directory](#) (i.e. `~/poky/build` in this example).

Also, since you are preparing to work on the kernel image, you need to set the [MACHINE\\_ESSENTIAL\\_EXTRA\\_RECOMMENDS](#) variable to include kernel modules.

This example uses the default "qemux86" for the `MACHINE` variable but needs to add the "kernel-modules":

```
MACHINE_ESSENTIAL_EXTRA_RECOMMENDS += "kernel-modules"
```

3. **Create a Layer for Patches:** You need to create a layer to hold patches created for the kernel image. You can use the `bitbake-layers create-layer` command as follows:

```
$ cd ~/poky/build
$ bitbake-layers create-layer ../../meta-mylayer
NOTE: Starting bitbake server...
Add your new layer with 'bitbake-layers add-layer ../../meta-mylayer'
```

\$

## Note

For background information on working with common and BSP layers, see the "[Understanding and Creating Layers](#)" section in the Yocto Project Development Tasks Manual and the "[BSP Layers](#)" section in the Yocto Project Board Support (BSP) Developer's Guide, respectively. For information on how to use the `bitbake-layers create-layer` command to quickly set up a layer, see the "[Creating a General Layer Using the bitbake-layers Script](#)" section in the Yocto Project Development Tasks Manual.

4. **Inform the BitBake Build Environment About Your Layer:** As directed when you created your layer, you need to add the layer to the `BBLAYERS` variable in the `bblayers.conf` file as follows:

```
$ cd ~/poky/build
$ bitbake-layers add-layer ../../meta-my-layer
NOTE: Starting bitbake server...
$
```

5. **Build the Extensible SDK:** Use BitBake to build the extensible SDK specifically for use with images to be run using QEMU:

```
$ cd ~/poky/build
$ bitbake core-image-minimal -c populate_sdk_ext
```

Once the build finishes, you can find the SDK installer file (i.e. `*.sh` file) in the following directory:

```
~/poky/build/tmp/deploy/sdk
```

For this example, the installer file is named `poky-glibc-x86_64-core-image-minimal-i586-toolchain-ext-3.0.1.sh`

6. **Install the Extensible SDK:** Use the following command to install the SDK. For this example, install the SDK in the default `~/poky_sdk` directory:

```
$ cd ~/poky/build/tmp/deploy/sdk
$./poky-glibc-x86_64-core-image-minimal-i586-toolchain-ext-3.0.1.sh
Poky (Yocto Project Reference Distro) Extensible SDK installer version 3.0.1
=====
Enter target directory for SDK (default: ~/poky_sdk):
You are about to install the SDK to "/home/scottrif/poky_sdk". Proceed [Y/n]? Y
Extracting SDK.....done
Setting it up...
Extracting buildtools...
Preparing build system...
Parsing recipes: 100% |#####| Time: 0:00:52
Initializing tasks: 100% |#####| Time: 0:00:04
Checking sstate mirror object availability: 100% |#####| Time: 0:00:00
Parsing recipes: 100% |#####| Time: 0:00:33
Initializing tasks: 100% |#####| Time: 0:00:00
done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.
$. /home/scottrif/poky_sdk/environment-setup-i586-poky-linux
```

7. **Set Up a New Terminal to Work With the Extensible SDK:** You must set up a new terminal to work with the SDK. You cannot use the same BitBake shell used to build the installer.

After opening a new shell, run the SDK environment setup script as directed by the output from installing the SDK:

```
$ source ~/poky_sdk/environment-setup-i586-poky-linux
"SDK environment now set up; additionally you may now run devtool to perform development tasks.
Run devtool --help for further details.
```

## Note

If you get a warning about attempting to use the extensible SDK in an environment set up to run BitBake, you did not use a new shell.

8. **Build the Clean Image:** The final step in preparing to work on the kernel is to build an initial image using `devtool` in the new terminal you just set up and initialized for SDK work:

```
$ devtool build-image
Parsing recipes: 100% |#####| Time: 0:00:05
Parsing of 830 .bb files complete (0 cached, 830 parsed). 1299 targets, 47 skipped, 0 masked, 0 errors.
WARNING: No packages to add, building image core-image-minimal unmodified
Loading cache: 100% |#####| Time: 0:00:00
Loaded 1299 entries from dependency cache.
NOTE: Resolving any missing task queue dependencies
Initializing tasks: 100% |#####| Time: 0:00:07
Checking sstate mirror object availability: 100% |#####| Time: 0:00:00
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
NOTE: Tasks Summary: Attempted 2866 tasks of which 2604 didn't need to be rerun and all succeeded.
NOTE: Successfully built core-image-minimal. You can find output files in /home/scottrif/poky_sdk/tmp/deploy/image:
```

If you were building for actual hardware and not for emulation, you could flash the image to a USB stick on `/dev/sdd` and boot your device. For an example that uses a Minnowboard, see the [TipsAndTricks/KernelDevelopmentWithEsdk](#) Wiki page.

At this point you have set up to start making modifications to the kernel by using the extensible SDK. For a continued example, see the "[Using devtool to Patch the Kernel](#)" section.

### 15.1.2. Getting Ready for Traditional Kernel Development¶

Getting ready for traditional kernel development using the Yocto Project involves many of the same steps as described in the previous section. However, you need to establish a local copy of the kernel source since you will be editing these files.

Follow these steps to prepare to update the kernel image using traditional kernel development flow with the Yocto Project. Completing this procedure leaves you ready to make modifications to the kernel source as described in the "[Using Traditional Kernel Development to Patch the Kernel](#)" section:

1. **Initialize the BitBake Environment:** Before you can do anything using BitBake, you need to initialize the BitBake build environment by sourcing the build environment script (i.e. `oe-init-build-env`). Also, for this example, be sure that the local branch you have checked out for `poky` is the Yocto Project Zeus branch. If you need to checkout out the Zeus branch, see the "[Checking out by Branch in Poky](#)" section in the Yocto Project Development Tasks Manual.

```
$ cd ~/poky
$ git branch
master
* Zeus
$ source oe-init-build-env
```

#### Note

The previous commands assume the [Source Repositories](#) (i.e. `poky`) have been cloned using Git and the local repository is named "poky".

2. **Prepare Your `local.conf` File:** By default, the `MACHINE` variable is set to "qemux86", which is fine if you are building for the QEMU emulator in 32-bit mode. However, if you are not, you need to set the `MACHINE` variable appropriately in your `conf/local.conf` file found in the [Build Directory](#) (i.e. `~/poky/build` in this example).

Also, since you are preparing to work on the kernel image, you need to set the `MACHINE_ESSENTIAL_EXTRA_RECOMMENDS` variable to include kernel modules.

This example uses the default "qemux86" for the `MACHINE` variable but needs to add the "kernel-modules":

```
MACHINE_ESSENTIAL_EXTRA_RECOMMENDS += "kernel-modules"
```

3. **Create a Layer for Patches:** You need to create a layer to hold patches created for the kernel image. You can use the `bitbake-layers create-layer` command as follows:

```
$ cd ~/poky/build
$ bitbake-layers create-layer ../../meta-my-layer
NOTE: Starting bitbake server...
Add your new layer with 'bitbake-layers add-layer ../../meta-my-layer'
```

#### Note

For background information on working with common and BSP layers, see the "[Understanding and Creating Layers](#)" section in the Yocto Project Development Tasks Manual and the "[BSP Layers](#)" section in the Yocto Project Board Support (BSP) Developer's Guide, respectively. For information on how to use the `bitbake-layers create-layer` command to quickly set up a layer, see the "[Creating a General Layer Using the bitbake-layers Script](#)" section in the Yocto Project Development Tasks Manual.

4. **Inform the BitBake Build Environment About Your Layer:** As directed when you created your layer, you need to add the layer to the `BBLAYERS` variable in the `bblayers.conf` file as follows:

```
$ cd ~/poky/build
$ bitbake-layers add-layer ../../meta-my-layer
NOTE: Starting bitbake server ...
$
```

5. **Create a Local Copy of the Kernel Git Repository:** You can find Git repositories of supported Yocto Project kernels organized under "Yocto Linux Kernel" in the Yocto Project Source Repositories at <http://git.yoctoproject.org>.

For simplicity, it is recommended that you create your copy of the kernel Git repository outside of the [Source Directory](#), which is usually named `poky`. Also, be sure you are in the `standard/base` branch.

The following commands show how to create a local copy of the `linux-yocto-4.12` kernel and be in the `standard/base` branch.

#### Note

The `linux-yocto-4.12` kernel can be used with the Yocto Project 2.4 release and forward. You cannot use the `linux-yocto-4.12` kernel with releases prior to Yocto Project 2.4:

```
$ cd ~
$ git clone git://git.yoctoproject.org/linux-yocto-4.12 --branch standard/base
Cloning into 'linux-yocto-4.12'...
remote: Counting objects: 6097195, done.
remote: Compressing objects: 100% (901026/901026), done.
remote: Total 6097195 (delta 5152604), reused 6096847 (delta 5152256)
Receiving objects: 100% (6097195/6097195), 1.24 GiB | 7.81 MiB/s, done.
Resolving deltas: 100% (5152604/5152604), done.
Checking connectivity... done.
Checking out files: 100% (59846/59846), done.
```

6. **Create a Local Copy of the Kernel Cache Git Repository:** For simplicity, it is recommended that you create your copy of the kernel cache Git repository outside of the [Source Directory](#), which is usually named `poky`. Also, for this example, be sure you are in the `yocto-4.12` branch.

The following commands show how to create a local copy of the `yocto-kernel-cache` and be in the `yocto-4.12` branch:

```
$ cd ~
$ git clone git://git.yoctoproject.org/yocto-kernel-cache --branch yocto-4.12
Cloning into 'yocto-kernel-cache'...
remote: Counting objects: 22639, done.
remote: Compressing objects: 100% (9761/9761), done.
remote: Total 22639 (delta 12400), reused 22586 (delta 12347)
Receiving objects: 100% (22639/22639), 22.34 MiB | 6.27 MiB/s, done.
Resolving deltas: 100% (12400/12400), done.
Checking connectivity... done.
```

At this point, you are ready to start making modifications to the kernel using traditional kernel development steps. For a continued example, see the ["Using Traditional Kernel Development to Patch the Kernel"](#) section.

## 15.2. Creating and Preparing a Layer¶

If you are going to be modifying kernel recipes, it is recommended that you create and prepare your own layer in which to do your work. Your layer contains its own [BitBake](#) append files ( `.bbappend`) and provides a convenient mechanism to create your own recipe files ( `.bb`) as well as store and use kernel patch files. For background information on working with layers, see the ["Understanding and Creating Layers"](#) section in the Yocto Project Development Tasks Manual.

### Tip

The Yocto Project comes with many tools that simplify tasks you need to perform. One such tool is the `bitbake-layers create-layer` command, which simplifies creating a new layer. See the ["Creating a General Layer Using the bitbake-layers Script"](#) section in the Yocto Project Development Tasks Manual for information on how to use this script to quick set up a new layer.

To better understand the layer you create for kernel development, the following section describes how to create a layer without the aid of tools. These steps assume creation of a layer named `mylayer` in your home directory:

1. **Create Structure:** Create the layer's structure:

```
$ cd $HOME
$ mkdir meta-mylayer
$ mkdir meta-mylayer/conf
$ mkdir meta-mylayer/recipes-kernel
$ mkdir meta-mylayer/recipes-kernel/linux
$ mkdir meta-mylayer/recipes-kernel/linux/linux-yocto
```

The `conf` directory holds your configuration files, while the `recipes-kernel` directory holds your append file and eventual patch files.

2. **Create the Layer Configuration File:** Move to the `meta-mylayer/conf` directory and create the `layer.conf` file as follows:

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

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

BBFILE_COLLECTIONS += "mylayer"
BBFILE_PATTERN_mylayer = "^${LAYERDIR}/"
BBFILE_PRIORITY_mylayer = "5"
```

Notice `mylayer` as part of the last three statements.

3. **Create the Kernel Recipe Append File:** Move to the `meta-mylayer/recipes-kernel/linux` directory and create the kernel's append file. This example uses the `linux-yocto-4.12` kernel. Thus, the name of the append file is `linux-yocto_4.12.bbappend`:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"

SRC_URI_append = " file://patch-file-one"
SRC_URI_append = " file://patch-file-two"
SRC_URI_append = " file://patch-file-three"
```

The [FILESEXTRAPATHS](#) and [SRC\\_URI](#) statements enable the OpenEmbedded build system to find patch files. For more information on using append files, see the ["Using .bbappend Files in Your Layer"](#) section in the Yocto Project Development Tasks

### 15.3. Modifying an Existing Recipe¶

In many cases, you can customize an existing linux-yocto recipe to meet the needs of your project. Each release of the Yocto Project provides a few Linux kernel recipes from which you can choose. These are located in the [Source Directory](#) in meta/recipes-kernel/linux.

Modifying an existing recipe can consist of the following:

- Creating the append file
- Applying patches
- Changing the configuration

Before modifying an existing recipe, be sure that you have created a minimal, custom layer from which you can work. See the ["Creating and Preparing a Layer"](#) section for information.

#### 15.3.1. Creating the Append File¶

You create this file in your custom layer. You also name it accordingly based on the linux-yocto recipe you are using. For example, if you are modifying the meta/recipes-kernel/linux/linux-yocto\_4.12.bb recipe, the append file will typically be located as follows within your custom layer:

```
your-Layer/recipes-kernel/linux/linux-yocto_4.12.bbappend
```

The append file should initially extend the [FILES\\_PATH](#) search path by prepending the directory that contains your files to the [FILESEXTRAPATHS](#) variable as follows:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
```

The path `${THISDIR}/${PN}` expands to "linux-yocto" in the current directory for this example. If you add any new files that modify the kernel recipe and you have extended [FILES\\_PATH](#) as described above, you must place the files in your layer in the following area:

```
your-Layer/recipes-kernel/linux/linux-yocto/
```

#### Note

If you are working on a new machine Board Support Package (BSP), be sure to refer to the Yocto Project Board Support Package (BSP) Developer's Guide.

As an example, consider the following append file used by the BSPs in meta-yocto-bsp:

```
meta-yocto-bsp/recipes-kernel/linux/linux-yocto_4.12.bbappend
```

The following listing shows the file. Be aware that the actual commit ID strings in this example listing might be different than the actual strings in the file from the meta-yocto-bsp layer upstream.

```
KBRANCH_genericx86 = "standard/base"
KBRANCH_genericx86-64 = "standard/base"

KMACHINE_genericx86 ?= "common-pc"
KMACHINE_genericx86-64 ?= "common-pc-64"
KBRANCH_edgerouter = "standard/edgerouter"
KBRANCH_beaglebone = "standard/beaglebone"
KBRANCH_mpc8315e-rdb = "standard/fsl-mpc8315e-rdb"

SRCREV_machine_genericx86 ?= "d09f2ce584d60ecb7890550c22a80c48b83c2e19"
SRCREV_machine_genericx86-64 ?= "d09f2ce584d60ecb7890550c22a80c48b83c2e19"
SRCREV_machine_edgerouter ?= "b5c8cfa2dfe296410d51e131289fb09c69e1e7d"
SRCREV_machine_beaglebone ?= "b5c8cfa2dfe296410d51e131289fb09c69e1e7d"
SRCREV_machine_mpc8315e-rdb ?= "2d1d010240846d7bff15d1fcc0cb6eb8a22fc78a"

COMPATIBLE_MACHINE_genericx86 = "genericx86"
COMPATIBLE_MACHINE_genericx86-64 = "genericx86-64"
COMPATIBLE_MACHINE_edgerouter = "edgerouter"
COMPATIBLE_MACHINE_beaglebone = "beaglebone"
COMPATIBLE_MACHINE_mpc8315e-rdb = "mpc8315e-rdb"

LINUX_VERSION_genericx86 = "4.12.7"
LINUX_VERSION_genericx86-64 = "4.12.7"
LINUX_VERSION_edgerouter = "4.12.10"
LINUX_VERSION_beaglebone = "4.12.10"
LINUX_VERSION_mpc8315e-rdb = "4.12.10"
```

This append file contains statements used to support several BSPs that ship with the Yocto Project. The file defines machines using the [COMPATIBLE\\_MACHINE](#) variable and uses the [KMACHINE](#) variable to ensure the machine name used by the OpenEmbedded build system maps to the machine name used by the Linux Yocto kernel. The file also uses the optional [KBRANCH](#) variable to ensure the build process uses the appropriate kernel branch.

Although this particular example does not use it, the [KERNEL\\_FEATURES](#) variable could be used to enable features specific to the kernel. The append file points to specific commits in the [Source Directory](#) Git repository and the meta Git repository branches to identify the exact kernel needed to build the BSP.



One thing missing in this particular BSP, which you will typically need when developing a BSP, is the kernel configuration file (`.config`) for your BSP. When developing a BSP, you probably have a kernel configuration file or a set of kernel configuration files that, when taken together, define the kernel configuration for your BSP. You can accomplish this definition by putting the configurations in a file or a set of files inside a directory located at the same level as your kernel's append file and having the same name as the kernel's main recipe file. With all these conditions met, simply reference those files in the `SRC_URI` statement in the append file.

For example, suppose you had some configuration options in a file called `network_configs.cfg`. You can place that file inside a directory named `linux-yocto` and then add a `SRC_URI` statement such as the following to the append file. When the OpenEmbedded build system builds the kernel, the configuration options are picked up and applied.

```
SRC_URI += "file://network_configs.cfg"
```

To group related configurations into multiple files, you perform a similar procedure. Here is an example that groups separate configurations specifically for Ethernet and graphics into their own files and adds the configurations by using a `SRC_URI` statement like the following in your append file:

```
SRC_URI += "file://myconfig.cfg \
 file://eth.cfg \
 file://gfx.cfg"
```

Another variable you can use in your kernel recipe append file is the `FILESEXTRAPATHS` variable. When you use this statement, you are extending the locations used by the OpenEmbedded system to look for files and patches as the recipe is processed.

## Note

Other methods exist to accomplish grouping and defining configuration options. For example, if you are working with a local clone of the kernel repository, you could checkout the kernel's `meta` branch, make your changes, and then push the changes to the local bare clone of the kernel. The result is that you directly add configuration options to the `meta` branch for your BSP. The configuration options will likely end up in that location anyway if the BSP gets added to the Yocto Project.

In general, however, the Yocto Project maintainers take care of moving the `SRC_URI`-specified configuration options to the kernel's `meta` branch. Not only is it easier for BSP developers to not have to worry about putting those configurations in the branch, but having the maintainers do it allows them to apply 'global' knowledge about the kinds of common configuration options multiple BSPs in the tree are typically using. This allows for promotion of common configurations into common features.

### 15.3.2. Applying Patches¶

If you have a single patch or a small series of patches that you want to apply to the Linux kernel source, you can do so just as you would with any other recipe. You first copy the patches to the path added to `FILESEXTRAPATHS` in your `.bbappend` file as described in the previous section, and then reference them in `SRC_URI` statements.

For example, you can apply a three-patch series by adding the following lines to your `linux-yocto .bbappend` file in your layer:

```
SRC_URI += "file://0001-first-change.patch"
SRC_URI += "file://0002-second-change.patch"
SRC_URI += "file://0003-third-change.patch"
```

The next time you run BitBake to build the Linux kernel, BitBake detects the change in the recipe and fetches and applies the patches before building the kernel.

For a detailed example showing how to patch the kernel using `devtool`, see the "[Using devtool to Patch the Kernel](#)" and "[Using Traditional Kernel Development to Patch the Kernel](#)" sections.

### 15.3.3. Changing the Configuration¶

You can make wholesale or incremental changes to the final `.config` file used for the eventual Linux kernel configuration by including a `defconfig` file and by specifying configuration fragments in the `SRC_URI` to be applied to that file.

If you have a complete, working Linux kernel `.config` file you want to use for the configuration, as before, copy that file to the appropriate `{PN}` directory in your layer's `recipes-kernel/linux` directory, and rename the copied file to "defconfig". Then, add the following lines to the `linux-yocto .bbappend` file in your layer:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
SRC_URI += "file://defconfig"
```

The `SRC_URI` tells the build system how to search for the file, while the `FILESEXTRAPATHS` extends the `FILESPATH` variable (search directories) to include the `{PN}` directory you created to hold the configuration changes.

## Note

The build system applies the configurations from the `defconfig` file before applying any subsequent configuration fragments. The final kernel configuration is a combination of the configurations in the `defconfig` file and any configuration fragments you provide. You need to realize that if you have any configuration fragments, the build system applies these on top of and after applying the existing `defconfig` file configurations.

Generally speaking, the preferred approach is to determine the incremental change you want to make and add that as a configuration fragment. For example, if you want to add support for a basic serial console, create a file named `8250.cfg` in the `{PN}` directory



with the following content (without indentation):

```
CONFIG_SERIAL_8250=y
CONFIG_SERIAL_8250_CONSOLE=y
CONFIG_SERIAL_8250_PCI=y
CONFIG_SERIAL_8250_NR_UARTS=4
CONFIG_SERIAL_8250_RUNTIME_UARTS=4
CONFIG_SERIAL_CORE=y
CONFIG_SERIAL_CORE_CONSOLE=y
```

Next, include this configuration fragment and extend the `FILESPATH` variable in your `.bbappend` file:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
SRC_URI += "file://8250.cfg"
```

The next time you run BitBake to build the Linux kernel, BitBake detects the change in the recipe and fetches and applies the new configuration before building the kernel.

For a detailed example showing how to configure the kernel, see the "[Configuring the Kernel](#)" section.

### 15.3.4. Using an "In-Tree" `defconfig` File¶

It might be desirable to have kernel configuration fragment support through a `defconfig` file that is pulled from the kernel source tree for the configured machine. By default, the OpenEmbedded build system looks for `defconfig` files in the layer used for Metadata, which is "out-of-tree", and then configures them using the following:

```
SRC_URI += "file://defconfig"
```

If you do not want to maintain copies of `defconfig` files in your layer but would rather allow users to use the default configuration from the kernel tree and still be able to add configuration fragments to the `SRC_URI` through, for example, append files, you can direct the OpenEmbedded build system to use a `defconfig` file that is "in-tree".

To specify an "in-tree" `defconfig` file, use the following statement form:

```
KBUILD_DEFCONFIG_KMACHINE ?= defconfig_file
```

Here is an example that appends the `KBUILD_DEFCONFIG` variable with "common-pc" and provides the path to the "in-tree" `defconfig` file:

```
KBUILD_DEFCONFIG_common-pc ?= "/home/scottrif/configfiles/my_defconfig_file"
```

Aside from modifying your kernel recipe and providing your own `defconfig` file, you need to be sure no files or statements set `SRC_URI` to use a `defconfig` other than your "in-tree" file (e.g. a kernel's `linux-machine.inc` file). In other words, if the build system detects a statement that identifies an "out-of-tree" `defconfig` file, that statement will override your `KBUILD_DEFCONFIG` variable.

See the [KBUILD\\_DEFCONFIG](#) variable description for more information.

## 15.4. Using `devtool` to Patch the Kernel¶

The steps in this procedure show you how you can patch the kernel using the extensible SDK and `devtool`.

### Note

Before attempting this procedure, be sure you have performed the steps to get ready for updating the kernel as described in the "[Getting Ready to Develop Using `devtool`](#)" section.

Patching the kernel involves changing or adding configurations to an existing kernel, changing or adding recipes to the kernel that are needed to support specific hardware features, or even altering the source code itself.

This example creates a simple patch by adding some QEMU emulator console output at boot time through `printk` statements in the kernel's `calibrate.c` source code file. Applying the patch and booting the modified image causes the added messages to appear on the emulator's console. The example is a continuation of the setup procedure found in the "[Getting Ready to Develop Using `devtool`](#)" Section.

1. **Check Out the Kernel Source Files:** First you must use `devtool` to checkout the kernel source code in its workspace. Be sure you are in the terminal set up to do work with the extensible SDK.

### Note

See this [step](#) in the "[Getting Ready to Develop Using `devtool`](#)" section for more information.

Use the following `devtool` command to check out the code:

```
$ devtool modify linux-yocto
```

### Note

During the checkout operation, a bug exists that could cause errors such as the following to appear:

```
ERROR: Taskhash mismatch 2c793438c2d9f8c3681fd5f7bc819efa versus
be3a89ce7c47178880ba7bf6293d7404 for
/path/to/esdk/layers/poky/meta/recipes-kernel/linux/linux-yocto_4.10.bb.do_unpack
```

You can safely ignore these messages. The source code is correctly checked out.

2. **Edit the Source Files** Follow these steps to make some simple changes to the source files:

- a. **Change the working directory:** In the previous step, the output noted where you can find the source files (e.g. `~/poky_sdk/workspace/sources/linux-yocto`). Change to where the kernel source code is before making your edits to the `calibrate.c` file:

```
$ cd ~/poky_sdk/workspace/sources/linux-yocto
```

- b. **Edit the source file:** Edit the `init/calibrate.c` file to have the following changes:

```
void calibrate_delay(void)
{
 unsigned long lpj;
 static bool printed;
 int this_cpu = smp_processor_id();

 printk("*****\n");
 printk("* *\n");
 printk("* HELLO YOCTO KERNEL *\n");
 printk("* *\n");
 printk("*****\n");

 if (per_cpu(cpu_loops_per_jiffy, this_cpu)) {
 .
 .
 .
 }
}
```

3. **Build the Updated Kernel Source:** To build the updated kernel source, use `devtool`:

```
$ devtool build linux-yocto
```

4. **Create the Image With the New Kernel:** Use the `devtool build-image` command to create a new image that has the new kernel.

## Note

If the image you originally created resulted in a Wic file, you can use an alternate method to create the new image with the updated kernel. For an example, see the steps in the [TipsAndTricks/KernelDevelopmentWithEsdk](#) Wiki Page.

```
$ cd ~
$ devtool build-image core-image-minimal
```

5. **Test the New Image:** For this example, you can run the new image using QEMU to verify your changes:

- a. **Boot the image:** Boot the modified image in the QEMU emulator using this command:

```
$ runqemu qemux86
```

- b. **Verify the changes:** Log into the machine using `root` with no password and then use the following shell command to scroll through the console's boot output.

```
dmesg | less
```

You should see the results of your `printk` statements as part of the output when you scroll down the console window.

6. **Stage and commit your changes:** Within your eSDK terminal, change your working directory to where you modified the `calibrate.c` file and use these Git commands to stage and commit your changes:

```
$ cd ~/poky_sdk/workspace/sources/linux-yocto
$ git status
$ git add init/calibrate.c
$ git commit -m "calibrate: Add printk example"
```

7. **Export the Patches and Create an Append File:** To export your commits as patches and create a `.bbappend` file, use the following command in the terminal used to work with the extensible SDK. This example uses the previously established layer named `meta-mylayer`.

## Note

See Step 3 of the ["Getting Ready to Develop Using devtool"](#) section for information on setting up this layer.

```
$ devtool finish linux-yocto ~/meta-mylayer
```

Once the command finishes, the patches and the `.bbappend` file are located in the `~/meta-mylayer/recipes-kernel/linux` directory.

8. **Build the Image With Your Modified Kernel:** You can now build an image that includes your kernel patches. Execute the following command from your [Build Directory](#) in the terminal set up to run BitBake:

```
$ cd ~/poky/build
$ bitbake core-image-minimal
```

## 15.5. Using Traditional Kernel Development to Patch the Kernel¶

The steps in this procedure show you how you can patch the kernel using traditional kernel development (i.e. not using `devtool` and the extensible SDK as described in the ["Using devtool to Patch the Kernel"](#) section).

### Note

Before attempting this procedure, be sure you have performed the steps to get ready for updating the kernel as described in the ["Getting Ready for Traditional Kernel Development"](#) section.

Patching the kernel involves changing or adding configurations to an existing kernel, changing or adding recipes to the kernel that are needed to support specific hardware features, or even altering the source code itself.

The example in this section creates a simple patch by adding some QEMU emulator console output at boot time through `printk` statements in the kernel's `calibrate.c` source code file. Applying the patch and booting the modified image causes the added messages to appear on the emulator's console. The example is a continuation of the setup procedure found in the ["Getting Ready for Traditional Kernel Development"](#) Section.

1. **Edit the Source Files** Prior to this step, you should have used Git to create a local copy of the repository for your kernel. Assuming you created the repository as directed in the ["Getting Ready for Traditional Kernel Development"](#) section, use the following commands to edit the `calibrate.c` file:

- a. **Change the working directory:** You need to locate the source files in the local copy of the kernel Git repository: Change to where the kernel source code is before making your edits to the `calibrate.c` file:

```
$ cd ~/linux-yocto-4.12/init
```

- b. **Edit the source file:** Edit the `calibrate.c` file to have the following changes:

```
void calibrate_delay(void)
{
 unsigned long lpj;
 static bool printed;
 int this_cpu = smp_processor_id();

 printk("*****\n");
 printk("*\n");
 printk(" HELLO YOCTO KERNEL *\n");
 printk("*\n");
 printk("*****\n");

 if (per_cpu(cpu_loops_per_jiffy, this_cpu)) {
 .
 .
 .
 }
}
```

2. **Stage and Commit Your Changes:** Use standard Git commands to stage and commit the changes you just made:

```
$ git add calibrate.c
$ git commit -m "calibrate.c - Added some printk statements"
```

If you do not stage and commit your changes, the OpenEmbedded Build System will not pick up the changes.

3. **Update Your `local.conf` File to Point to Your Source Files:** In addition to your `local.conf` file specifying to use "kernel-modules" and the "qemux86" machine, it must also point to the updated kernel source files. Add [SRC\\_URI](#) and [SRCREV](#) statements similar to the following to your `local.conf`:

```
$ cd ~/poky/build/conf
```

Add the following to the `local.conf`:

```
SRC_URI_pn-linux-yocto = "git://path-to/linux-yocto-4.12;protocol=file;name=machine;branch=standard/base; \
 git://path-to/yocto-kernel-cache;protocol=file;type=kmeta;name=meta;branch=yocto-4.12;de
SRCREV_meta_qemux86 = "${AUTOREV}"
SRCREV_machine_qemux86 = "${AUTOREV}"
```

### Note

Be sure to replace `path-to` with the pathname to your local Git repositories. Also, you must be sure to specify the correct branch and machine types. For this example, the branch is `standard/base` and the machine is "qemux86".

4. **Build the Image:** With the source modified, your changes staged and committed, and the `local.conf` file pointing to the kernel files, you can now use BitBake to build the image:

```
$ cd ~/poky/build
$ bitbake core-image-minimal
```

5. **Boot the image:** Boot the modified image in the QEMU emulator using this command. When prompted to login to the QEMU console, use "root" with no password:

```
$ cd ~/poky/build
$ runqemu qemu86
```

6. **Look for Your Changes:** As QEMU booted, you might have seen your changes rapidly scroll by. If not, use these commands to see your changes:

```
dmesg | less
```

You should see the results of your `printk` statements as part of the output when you scroll down the console window.

7. **Generate the Patch File:** Once you are sure that your patch works correctly, you can generate a `*.patch` file in the kernel source repository:

```
$ cd ~/linux-yocto-4.12/init
$ git format-patch -1
0001-calibrate.c-Added-some-printk-statements.patch
```

8. **Move the Patch File to Your Layer:** In order for subsequent builds to pick up patches, you need to move the patch file you created in the previous step to your layer `meta-mylayer`. For this example, the layer created earlier is located in your home directory as `meta-mylayer`. When the layer was created using the `yocto-create` script, no additional hierarchy was created to support patches. Before moving the patch file, you need to add additional structure to your layer using the following commands:

```
$ cd ~/meta-mylayer
$ mkdir recipes-kernel
$ mkdir recipes-kernel/linux
$ mkdir recipes-kernel/linux/linux-yocto
```

Once you have created this hierarchy in your layer, you can move the patch file using the following command:

```
$ mv ~/linux-yocto-4.12/init/0001-calibrate.c-Added-some-printk-statements.patch ~/meta-mylayer/recipes-kernel/linux
```

9. **Create the Append File:** Finally, you need to create the `linux-yocto_4.12.bbappend` file and insert statements that allow the OpenEmbedded build system to find the patch. The append file needs to be in your layer's `recipes-kernel/linux` directory and it must be named `linux-yocto_4.12.bbappend` and have the following contents:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"

SRC_URI_append = " file://0001-calibrate.c-Added-some-printk-statements.patch"
```

The `FILESEXTRAPATHS` and `SRC_URI` statements enable the OpenEmbedded build system to find the patch file.

For more information on append files and patches, see the "[Creating the Append File](#)" and "[Applying Patches](#)" sections. You can also see the "[Using .bbappend Files in Your Layer](#)" section in the Yocto Project Development Tasks Manual.

## Note

To build `core-image-minimal` again and see the effects of your patch, you can essentially eliminate the temporary source files saved in `poky/build/tmp/work/. . .` and residual effects of the build by entering the following sequence of commands:

```
$ cd ~/poky/build
$ bitbake -c cleanall yocto-linux
$ bitbake core-image-minimal -c cleanall
$ bitbake core-image-minimal
$ runqemu qemu86
```

## 15.6. Configuring the Kernel

Configuring the Yocto Project kernel consists of making sure the `.config` file has all the right information in it for the image you are building. You can use the `menuconfig` tool and configuration fragments to make sure your `.config` file is just how you need it. You can also save known configurations in a `defconfig` file that the build system can use for kernel configuration.

This section describes how to use `menuconfig`, create and use configuration fragments, and how to interactively modify your `.config` file to create the leanest kernel configuration file possible.

For more information on kernel configuration, see the "[Changing the Configuration](#)" section.

### 15.6.1. Using menuconfig

The easiest way to define kernel configurations is to set them through the `menuconfig` tool. This tool provides an interactive method with which to set kernel configurations. For general information on `menuconfig`, see <http://en.wikipedia.org/wiki/Menuconfig>.

To use the `menuconfig` tool in the Yocto Project development environment, you must do the following:

- Because you launch `menuconfig` using BitBake, you must be sure to set up your environment by running the `oe-init-build-env` script found in the [Build Directory](#).
- You must be sure of the state of your build's configuration in the [Source Directory](#).
- Your build host must have the following two packages installed:

```
libncurses5-dev
libtinfo-dev
```

The following commands initialize the BitBake environment, run the `do_kernel_configme` task, and launch `menuconfig`. These commands assume the Source Directory's top-level folder is `~/poky`:

```
$ cd poky
$ source oe-init-build-env
$ bitbake linux-yocto -c kernel_configme -f
$ bitbake linux-yocto -c menuconfig
```

Once `menuconfig` comes up, its standard interface allows you to interactively examine and configure all the kernel configuration parameters. After making your changes, simply exit the tool and save your changes to create an updated version of the `.config` configuration file.

## Note

You can use the entire `.config` file as the `defconfig` file. For information on `defconfig` files, see the ["Changing the Configuration"](#), ["Using an In-Tree defconfig File"](#), and ["Creating a defconfig File"](#) sections.

Consider an example that configures the `"CONFIG_SMP"` setting for the `linux-yocto-4.12` kernel.

## Note

The OpenEmbedded build system recognizes this kernel as `linux-yocto` through Metadata (e.g. `PREFERRED_VERSION_linux-yocto ?= "12.4%"`).

Once `menuconfig` launches, use the interface to navigate through the selections to find the configuration settings in which you are interested. For this example, you deselect `"CONFIG_SMP"` by clearing the "Symmetric Multi-Processing Support" option. Using the interface, you can find the option under "Processor Type and Features". To deselect `"CONFIG_SMP"`, use the arrow keys to highlight "Symmetric Multi-Processing Support" and enter "N" to clear the asterisk. When you are finished, exit out and save the change.

Saving the selections updates the `.config` configuration file. This is the file that the OpenEmbedded build system uses to configure the kernel during the build. You can find and examine this file in the Build Directory in `tmp/work/`. The actual `.config` is located in the area where the specific kernel is built. For example, if you were building a Linux Yocto kernel based on the `linux-yocto-4.12` kernel and you were building a QEMU image targeted for x86 architecture, the `.config` file would be:

```
poky/build/tmp/work/qemux86-poky-linux/linux-yocto/4.12.12+gitAUTOINC+eda4d18...
...967-r0/linux-qemux86-standard-build/.config
```

## Note

The previous example directory is artificially split and many of the characters in the actual filename are omitted in order to make it more readable. Also, depending on the kernel you are using, the exact pathname might differ.

Within the `.config` file, you can see the kernel settings. For example, the following entry shows that symmetric multi-processor support is not set:

```
CONFIG_SMP is not set
```

A good method to isolate changed configurations is to use a combination of the `menuconfig` tool and simple shell commands. Before changing configurations with `menuconfig`, copy the existing `.config` and rename it to something else, use `menuconfig` to make as many changes as you want and save them, then compare the renamed configuration file against the newly created file. You can use the resulting differences as your base to create configuration fragments to permanently save in your kernel layer.

## Note

Be sure to make a copy of the `.config` file and do not just rename it. The build system needs an existing `.config` file from which to work.

### 15.6.2. Creating a defconfig File¶

A `defconfig` file in the context of the Yocto Project is often a `.config` file that is copied from a build or a `defconfig` taken from the kernel tree and moved into recipe space. You can use a `defconfig` file to retain a known set of kernel configurations from which the OpenEmbedded build system can draw to create the final `.config` file.

## Note

Out-of-the-box, the Yocto Project never ships a `defconfig` or `.config` file. The OpenEmbedded build system creates the final `.config` file used to configure the kernel.

To create a `defconfig`, start with a complete, working Linux kernel `.config` file. Copy that file to the appropriate `${PN}` directory in your layer's `recipes-kernel/linux` directory, and rename the copied file to "defconfig" (e.g. `~/meta-my-layer/recipes-kernel/linux/linux-yocto/defconfig`). Then, add the following lines to the `linux-yocto.bbappend` file in your layer:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
SRC_URI += "file://defconfig"
```

The `SRC_URI` tells the build system how to search for the file, while the `FILESEXTRAPATHS` extends the `FILESPATH` variable (search directories) to include the `${PN}` directory you created to hold the configuration changes.

## Note

The build system applies the configurations from the `defconfig` file before applying any subsequent configuration fragments. The final kernel configuration is a combination of the configurations in the `defconfig` file and any configuration fragments you provide. You need to realize that if you have any configuration fragments, the build system applies these on top of and after applying the existing `defconfig` file configurations.

For more information on configuring the kernel, see the "[Changing the Configuration](#)" section.

### 15.6.3. Creating Configuration Fragments

Configuration fragments are simply kernel options that appear in a file placed where the OpenEmbedded build system can find and apply them. The build system applies configuration fragments after applying configurations from a `defconfig` file. Thus, the final kernel configuration is a combination of the configurations in the `defconfig` file and then any configuration fragments you provide. The build system applies fragments on top of and after applying the existing `defconfig` file configurations.

Syntactically, the configuration statement is identical to what would appear in the `.config` file, which is in the [Build Directory](#).

## Note

For more information about where the `.config` file is located, see the example in the "[Using menuconfig](#)" section.

It is simple to create a configuration fragment. One method is to use shell commands. For example, issuing the following from the shell creates a configuration fragment file named `my_smp.cfg` that enables multi-processor support within the kernel:

```
$ echo "CONFIG_SMP=y" >> my_smp.cfg
```

## Note

All configuration fragment files must use the `.cfg` extension in order for the OpenEmbedded build system to recognize them as a configuration fragment.

Another method is to create a configuration fragment using the differences between two configuration files: one previously created and saved, and one freshly created using the `menuconfig` tool.

To create a configuration fragment using this method, follow these steps:

1. **Complete a Build Through Kernel Configuration:** Complete a build at least through the kernel configuration task as follows:

```
$ bitbake linux-yocto -c kernel_configme -f
```

This step ensures that you create a `.config` file from a known state. Because situations exist where your build state might become unknown, it is best to run this task prior to starting `menuconfig`.

2. **Launch menuconfig:** Run the `menuconfig` command:

```
$ bitbake linux-yocto -c menuconfig
```

3. **Create the Configuration Fragment:** Run the `diffconfig` command to prepare a configuration fragment. The resulting file `fragment.cfg` is placed in the `${WORKDIR}` directory:

```
$ bitbake linux-yocto -c diffconfig
```

The `diffconfig` command creates a file that is a list of Linux kernel `CONFIG_` assignments. See the "[Changing the Configuration](#)" section for additional information on how to use the output as a configuration fragment.

**Note**

You can also use this method to create configuration fragments for a BSP. See the "[BSP Descriptions](#)" section for more information.

Where do you put your configuration fragment files? You can place these files in an area pointed to by [SRC\\_URI](#) as directed by your `bblayers.conf` file, which is located in your layer. The OpenEmbedded build system picks up the configuration and adds it to the kernel's configuration. For example, suppose you had a set of configuration options in a file called `myconfig.cfg`. If you put that file inside a directory named `linux-yocto` that resides in the same directory as the kernel's append file within your layer and then add the following statements to the kernel's append file, those configuration options will be picked up and applied when the kernel is built:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
SRC_URI += "file://myconfig.cfg"
```

As mentioned earlier, you can group related configurations into multiple files and name them all in the `SRC_URI` statement as well. For example, you could group separate configurations specifically for Ethernet and graphics into their own files and add those by using a `SRC_URI` statement like the following in your append file:

```
SRC_URI += "file://myconfig.cfg \
file://eth.cfg \
file://gfx.cfg"
```

**15.6.4. Validating Configuration**

You can use the `do_kernel_configcheck` task to provide configuration validation:

```
$ bitbake linux-yocto -c kernel_configcheck -f
```

Running this task produces warnings for when a requested configuration does not appear in the final `.config` file or when you override a policy configuration in a hardware configuration fragment.

In order to run this task, you must have an existing `.config` file. See the "[Using menuconfig](#)" section for information on how to create a configuration file.

Following is sample output from the `do_kernel_configcheck` task:

```
Loading cache: 100% |#####| Time: 0:00:00
Loaded 1275 entries from dependency cache.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
.
.
.

NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
WARNING: linux-yocto-4.12.12+gitAUTOINC+eda4d18ce4_16de014967-r0 do_kernel_configcheck:
[kernel config]: specified values did not make it into the kernel's final configuration:

----- CONFIG_X86_TSC -----
Config: CONFIG_X86_TSC
From: /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/bsp/common-pc/common-
Requested value: CONFIG_X86_TSC=y
Actual value:

----- CONFIG_X86_BIGSMP -----
Config: CONFIG_X86_BIGSMP
From: /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/cfg/smp.cfg
/home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/defconfig
Requested value: # CONFIG_X86_BIGSMP is not set
Actual value:

----- CONFIG_NR_CPUS -----
Config: CONFIG_NR_CPUS
From: /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/cfg/smp.cfg
/home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/bsp/common-pc/common-
/home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/defconfig
Requested value: CONFIG_NR_CPUS=8
Actual value: CONFIG_NR_CPUS=1

----- CONFIG_SCHED_SMT -----
Config: CONFIG_SCHED_SMT
From: /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/cfg/smp.cfg
/home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/defconfig
Requested value: CONFIG_SCHED_SMT=y
Actual value:

NOTE: Tasks Summary: Attempted 288 tasks of which 285 didn't need to be rerun and all succeeded.

Summary: There were 3 WARNING messages shown.
```

**Note**



The previous output example has artificial line breaks to make it more readable.

The output describes the various problems that you can encounter along with where to find the offending configuration items. You can use the information in the logs to adjust your configuration files and then repeat the `do_kernel_configme` and `do_kernel_configcheck` tasks until they produce no warnings.

For more information on how to use the `menuconfig` tool, see the "[Using menuconfig](#)" section.

### 15.6.5. Fine-Tuning the Kernel Configuration File¶

You can make sure the `.config` file is as lean or efficient as possible by reading the output of the kernel configuration fragment audit, noting any issues, making changes to correct the issues, and then repeating.

As part of the kernel build process, the `do_kernel_configcheck` task runs. This task validates the kernel configuration by checking the final `.config` file against the input files. During the check, the task produces warning messages for the following issues:

- Requested options that did not make the final `.config` file.
- Configuration items that appear twice in the same configuration fragment.
- Configuration items tagged as "required" that were overridden.
- A board overrides a non-board specific option.
- Listed options not valid for the kernel being processed. In other words, the option does not appear anywhere.

#### Note

The `do_kernel_configcheck` task can also optionally report if an option is overridden during processing.

For each output warning, a message points to the file that contains a list of the options and a pointer to the configuration fragment that defines them. Collectively, the files are the key to streamlining the configuration.

To streamline the configuration, do the following:

1. **Use a Working Configuration:** Start with a full configuration that you know works. Be sure the configuration builds and boots successfully. Use this configuration file as your baseline.
2. **Run Configure and Check Tasks:** Separately run the `do_kernel_configme` and `do_kernel_configcheck` tasks:
 

```
$ bitbake linux-yocto -c kernel_configme -f
$ bitbake linux-yocto -c kernel_configcheck -f
```
3. **Process the Results:** Take the resulting list of files from the `do_kernel_configcheck` task warnings and do the following:
  - Drop values that are redefined in the fragment but do not change the final `.config` file.
  - Analyze and potentially drop values from the `.config` file that override required configurations.
  - Analyze and potentially remove non-board specific options.
  - Remove repeated and invalid options.
4. **Re-Run Configure and Check Tasks:** After you have worked through the output of the kernel configuration audit, you can re-run the `do_kernel_configme` and `do_kernel_configcheck` tasks to see the results of your changes. If you have more issues, you can deal with them as described in the previous step.

Iteratively working through steps two through four eventually yields a minimal, streamlined configuration file. Once you have the best `.config`, you can build the Linux Yocto kernel.

### 15.7. Expanding Variables¶

Sometimes it is helpful to determine what a variable expands to during a build. You can do examine the values of variables by examining the output of the `bitbake -e` command. The output is long and is more easily managed in a text file, which allows for easy searches:

```
$ bitbake -e virtual/kernel > some_text_file
```

Within the text file, you can see exactly how each variable is expanded and used by the OpenEmbedded build system.

### 15.8. Working with a "Dirty" Kernel Version String¶

If you build a kernel image and the version string has a "+" or a "-dirty" at the end, uncommitted modifications exist in the kernel's source directory. Follow these steps to clean up the version string:

1. **Discover the Uncommitted Changes:** Go to the kernel's locally cloned Git repository (source directory) and use the following Git command to list the files that have been changed, added, or removed:

```
$ git status
```

2. **Commit the Changes:** You should commit those changes to the kernel source tree regardless of whether or not you will save, export, or use the changes:

```
$ git add
$ git commit -s -a -m "getting rid of -dirty"
```

3. **Rebuild the Kernel Image:** Once you commit the changes, rebuild the kernel.

Depending on your particular kernel development workflow, the commands you use to rebuild the kernel might differ. For information on building the kernel image when using `devtool`, see the "[Using `devtool` to Patch the Kernel](#)" section. For information on building the kernel image when using Bitbake, see the "[Using Traditional Kernel Development to Patch the Kernel](#)" section.

## 15.9. Working With Your Own Sources¶

If you cannot work with one of the Linux kernel versions supported by existing linux-yocto recipes, you can still make use of the Yocto Project Linux kernel tooling by working with your own sources. When you use your own sources, you will not be able to leverage the existing kernel [Metadata](#) and stabilization work of the linux-yocto sources. However, you will be able to manage your own Metadata in the same format as the linux-yocto sources. Maintaining format compatibility facilitates converging with linux-yocto on a future, mutually-supported kernel version.

To help you use your own sources, the Yocto Project provides a linux-yocto custom recipe (`linux-yocto-custom.bb`) that uses `kernel.org` sources and the Yocto Project Linux kernel tools for managing kernel Metadata. You can find this recipe in the poky Git repository of the Yocto Project [Source Repository](#) at:

```
poky/meta-skeleton/recipes-kernel/linux/linux-yocto-custom.bb
```

Here are some basic steps you can use to work with your own sources:

1. **Create a Copy of the Kernel Recipe:** Copy the `linux-yocto-custom.bb` recipe to your layer and give it a meaningful name. The name should include the version of the Yocto Linux kernel you are using (e.g. `linux-yocto-myproject_4.12.bb`, where "4.12" is the base version of the Linux kernel with which you would be working).
2. **Create a Directory for Your Patches:** In the same directory inside your layer, create a matching directory to store your patches and configuration files (e.g. `linux-yocto-myproject`).
3. **Ensure You Have Configurations:** Make sure you have either a `defconfig` file or configuration fragment files in your layer. When you use the `linux-yocto-custom.bb` recipe, you must specify a configuration. If you do not have a `defconfig` file, you can run the following:

```
$ make defconfig
```

After running the command, copy the resulting `.config` file to the `files` directory in your layer as "defconfig" and then add it to the `SRC_URI` variable in the recipe.

Running the `make defconfig` command results in the default configuration for your architecture as defined by your kernel. However, no guarantee exists that this configuration is valid for your use case, or that your board will even boot. This is particularly true for non-x86 architectures.

To use non-x86 `defconfig` files, you need to be more specific and find one that matches your board (i.e. for arm, you look in `arch/arm/configs` and use the one that is the best starting point for your board).

4. **Edit the Recipe:** Edit the following variables in your recipe as appropriate for your project:
  - `SRC_URI`: The `SRC_URI` should specify a Git repository that uses one of the supported Git fetcher protocols (i.e. `file`, `git`, `http`, and so forth). The `SRC_URI` variable should also specify either a `defconfig` file or some configuration fragment files. The skeleton recipe provides an example `SRC_URI` as a syntax reference.
  - `LINUX_VERSION`: The Linux kernel version you are using (e.g. "4.12").
  - `LINUX_VERSION_EXTENSION`: The Linux kernel `CONFIG_LOCALVERSION` that is compiled into the resulting kernel and visible through the `uname` command.
  - `SRCREV`: The commit ID from which you want to build.
  - `PR`: Treat this variable the same as you would in any other recipe. Increment the variable to indicate to the OpenEmbedded build system that the recipe has changed.
  - `PV`: The default `PV` assignment is typically adequate. It combines the `LINUX_VERSION` with the Source Control Manager (SCM) revision as derived from the `SRCPV` variable. The combined results are a string with the following form:

```
3.19.11+git1+68a635bf8dfb64b02263c1ac80c948647cc76d5f_1+218bd8d2022b9852c60d32f0d770931e3cf343e2
```

While lengthy, the extra verbosity in `PV` helps ensure you are using the exact sources from which you intend to build.

- `COMPATIBLE_MACHINE`: A list of the machines supported by your new recipe. This variable in the example recipe is set by default to a regular expression that matches only the empty string, `"(^$)"`. This default setting triggers an explicit build failure. You must change it to match a list of the machines that your new recipe supports. For example, to support the `qemu86` and `qemu86-64` machines, use the following form:

```
COMPATIBLE_MACHINE = "qemu86|qemu86-64"
```

5. **Customize Your Recipe as Needed:** Provide further customizations to your recipe as needed just as you would customize an existing linux-yocto recipe. See the "[Modifying an Existing Recipe](#)" section for information.

## 15.10. Working with Out-of-Tree Modules¶

This section describes steps to build out-of-tree modules on your target and describes how to incorporate out-of-tree modules in the build.

### 15.10.1. Building Out-of-Tree Modules on the Target¶

While the traditional Yocto Project development model would be to include kernel modules as part of the normal build process, you might find it useful to build modules on the target. This could be the case if your target system is capable and powerful enough to handle the necessary compilation. Before deciding to build on your target, however, you should consider the benefits of using a proper cross-development environment from your build host.

If you want to be able to build out-of-tree modules on the target, there are some steps you need to take on the target that is running your SDK image. Briefly, the `kernel-dev` package is installed by default on all `*.sdk` images and the `kernel-devsrc` package is installed on many of the `*.sdk` images. However, you need to create some scripts prior to attempting to build the out-of-tree modules on the target that is running that image.

Prior to attempting to build the out-of-tree modules, you need to be on the target as root and you need to change to the `/usr/src/kernel` directory. Next, make the scripts:

```
cd /usr/src/kernel
make scripts
```

Because all SDK image recipes include `dev-pkgs`, the `kernel-dev` packages will be installed as part of the SDK image and the `kernel-devsrc` packages will be installed as part of applicable SDK images. The SDK uses the scripts when building out-of-tree modules. Once you have switched to that directory and created the scripts, you should be able to build your out-of-tree modules on the target.

### 15.10.2. Incorporating Out-of-Tree Modules¶

While it is always preferable to work with sources integrated into the Linux kernel sources, if you need an external kernel module, the `hello-mod.bb` recipe is available as a template from which you can create your own out-of-tree Linux kernel module recipe.

This template recipe is located in the `poky` Git repository of the Yocto Project [Source Repository](#) at:

```
poky/meta-skeleton/recipes-kernel/hello-mod/hello-mod_0.1.bb
```

To get started, copy this recipe to your layer and give it a meaningful name (e.g. `mymodule_1.0.bb`). In the same directory, create a new directory named `files` where you can store any source files, patches, or other files necessary for building the module that do not come with the sources. Finally, update the recipe as needed for the module. Typically, you will need to set the following variables:

- `DESCRIPTION`
- `LICENSE*`
- `SRC_URI`
- `PV`

Depending on the build system used by the module sources, you might need to make some adjustments. For example, a typical module `Makefile` looks much like the one provided with the `hello-mod` template:

```
obj-m := hello.o

SRC := $(shell pwd)

all:
$(MAKE) -C $(KERNEL_SRC) M=$(SRC)

modules_install:
$(MAKE) -C $(KERNEL_SRC) M=$(SRC) modules_install
...
```

The important point to note here is the `KERNEL_SRC` variable. The `module` class sets this variable and the `KERNEL_PATH` variable to `${STAGING_KERNEL_DIR}` with the necessary Linux kernel build information to build modules. If your module `Makefile` uses a different variable, you might want to override the `do_compile` step, or create a patch to the `Makefile` to work with the more typical `KERNEL_SRC` or `KERNEL_PATH` variables.

After you have prepared your recipe, you will likely want to include the module in your images. To do this, see the documentation for the following variables in the Yocto Project Reference Manual and set one of them appropriately for your machine configuration file:

- `MACHINE_ESSENTIAL_EXTRA_RDEPENDS`
- `MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS`
- `MACHINE_EXTRA_RDEPENDS`
- `MACHINE_EXTRA_RRECOMMENDS`

Modules are often not required for boot and can be excluded from certain build configurations. The following allows for the most flexibility:

```
MACHINE_EXTRA_RRECOMMENDS += "kernel-module-mymodule"
```

The value is derived by appending the module filename without the `.ko` extension to the string "kernel-module-".

Because the variable is `RRECOMMENDS` and not a `RDEPENDS` variable, the build will not fail if this module is not available to include in the image.

## 15.11. Inspecting Changes and Commits¶

A common question when working with a kernel is: "What changes have been applied to this tree?" Rather than using "grep" across directories to see what has changed, you can use Git to inspect or search the kernel tree. Using Git is an efficient way to see what has changed in the tree.

### 15.11.1. What Changed in a Kernel?¶

Following are a few examples that show how to use Git commands to examine changes. These examples are by no means the only way to see changes.

#### Note

In the following examples, unless you provide a commit range, `kernel.org` history is blended with Yocto Project kernel changes. You can form ranges by using branch names from the kernel tree as the upper and lower commit markers with the Git commands. You can see the branch names through the web interface to the Yocto Project source repositories at <http://git.yoctoproject.org>.

To see a full range of the changes, use the `git whatchanged` command and specify a commit range for the branch (`commit..commit`).

Here is an example that looks at what has changed in the `emenlow` branch of the `linux-yocto-3.19` kernel. The lower commit range is the commit associated with the `standard/base` branch, while the upper commit range is the commit associated with the `standard/emenlow` branch.

```
$ git whatchanged origin/standard/base..origin/standard/emenlow
```

To see short, one line summaries of changes use the `git log` command:

```
$ git log --oneline origin/standard/base..origin/standard/emenlow
```

Use this command to see code differences for the changes:

```
$ git diff origin/standard/base..origin/standard/emenlow
```

Use this command to see the commit log messages and the text differences:

```
$ git show origin/standard/base..origin/standard/emenlow
```

Use this command to create individual patches for each change. Here is an example that that creates patch files for each commit and places them in your `Documents` directory:

```
$ git format-patch -o $HOME/Documents origin/standard/base..origin/standard/emenlow
```

### 15.11.2. Showing a Particular Feature or Branch Change¶

Tags in the Yocto Project kernel tree divide changes for significant features or branches. The `git show tag` command shows changes based on a tag. Here is an example that shows `systemtap` changes:

```
$ git show systemtap
```

You can use the `git branch --contains tag` command to show the branches that contain a particular feature. This command shows the branches that contain the `systemtap` feature:

```
$ git branch --contains systemtap
```

## 15.12. Adding Recipe-Space Kernel Features¶

You can add kernel features in the `recipe-space` by using the `KERNEL_FEATURES` variable and by specifying the feature's `.SCC` file path in the `SRC_URI` statement. When you add features using this method, the OpenEmbedded build system checks to be sure the features are present. If the features are not present, the build stops. Kernel features are the last elements processed for configuring and patching the kernel. Therefore, adding features in this manner is a way to enforce specific features are present and enabled without needing to do a full audit of any other layer's additions to the `SRC_URI` statement.

You add a kernel feature by providing the feature as part of the `KERNEL_FEATURES` variable and by providing the path to the feature's `.SCC` file, which is relative to the root of the kernel Metadata. The OpenEmbedded build system searches all forms of kernel Metadata on the `SRC_URI` statement regardless of whether the Metadata is in the "kernel-cache", system kernel Metadata, or a recipe-space Metadata (i.e. part of the kernel recipe). See the "[Kernel Metadata Location](#)" section for additional information.

When you specify the feature's `.SCC` file on the `SRC_URI` statement, the OpenEmbedded build system adds the directory of that `.SCC` file along with all its subdirectories to the kernel feature search path. Because subdirectories are searched, you can reference a single `.SCC` file in the `SRC_URI` statement to reference multiple kernel features.

Consider the following example that adds the "test.scc" feature to the build.

1. **Create the Feature File:** Create a `.SCC` file and locate it just as you would any other patch file, `.cfg` file, or fetcher item you specify in the `SRC_URI` statement.

## Notes

- You must add the directory of the `.SCC` file to the fetcher's search path in the same manner as you would add a `.patch` file.
- You can create additional `.SCC` files beneath the directory that contains the file you are adding. All subdirectories are searched during the build as potential feature directories.

Continuing with the example, suppose the "test.scc" feature you are adding has a `test.SCC` file in the following directory:

```
my_recipe
|
+-linux-yocto
|
+-test.cfg
+-test.scc
```

In this example, the `linux-yocto` directory has both the feature `test.SCC` file and a similarly named configuration fragment file `test.cfg`.

2. **Add the Feature File to `SRC_URI`:** Add the `.SCC` file to the recipe's `SRC_URI` statement:

```
SRC_URI_append = " file://test.scc"
```

The leading space before the path is important as the path is appended to the existing path.

3. **Specify the Feature as a Kernel Feature:** Use the `KERNEL_FEATURES` statement to specify the feature as a kernel feature:

```
KERNEL_FEATURES_append = " test.scc"
```

The OpenEmbedded build system processes the kernel feature when it builds the kernel.

## Note

If other features are contained below "test.scc", then their directories are relative to the directory containing the `test.SCC` file.

## Chapter 16. Working with Advanced Metadata (`yocto-kernel-cache`)

### [16.1. Overview](#)

### [16.2. Using Kernel Metadata in a Recipe](#)

### [16.3. Kernel Metadata Syntax](#)

#### [16.3.1. Configuration](#)

#### [16.3.2. Patches](#)

#### [16.3.3. Features](#)

#### [16.3.4. Kernel Types](#)

#### [16.3.5. BSP Descriptions](#)

### [16.4. Kernel Metadata Location](#)

#### [16.4.1. Recipe-Space Metadata](#)

#### [16.4.2. Metadata Outside the Recipe-Space](#)

### [16.5. Organizing Your Source](#)

#### [16.5.1. Encapsulating Patches](#)

#### [16.5.2. Machine Branches](#)

#### [16.5.3. Feature Branches](#)

### [16.6. SCC Description File Reference](#)

## 16.1. Overview

In addition to supporting configuration fragments and patches, the Yocto Project kernel tools also support rich [Metadata](#) that you can use to define complex policies and Board Support Package (BSP) support. The purpose of the Metadata and the tools that manage it is to help you manage the complexity of the configuration and sources used to support multiple BSPs and Linux kernel types.

Kernel Metadata exists in many places. One area in the Yocto Project [Source Repositories](#) is the `yocto-kernel-cache` Git repository. You can find this repository grouped under the "Yocto Linux Kernel" heading in the [Yocto Project Source Repositories](#).

Kernel development tools ("kern-tools") exist also in the Yocto Project Source Repositories under the "Yocto Linux Kernel" heading in the `yocto-kernel-tools` Git repository. The recipe that builds these tools is `meta/recipes-kernel/kern-tools/kern-tools-native_git.bb` in the [Source Directory](#) (e.g. poky).

## 16.2. Using Kernel Metadata in a Recipe

As mentioned in the introduction, the Yocto Project contains kernel Metadata, which is located in the `yocto-kernel-cache` Git repository. This Metadata defines Board Support Packages (BSPs) that correspond to definitions in `linux-yocto` recipes for corresponding BSPs. A BSP consists of an aggregation of kernel policy and enabled hardware-specific features. The BSP can be influenced from within the `linux-yocto` recipe.

## Note

A Linux kernel recipe that contains kernel Metadata (e.g. inherits from the `linux-yocto.inc` file) is said to be a "linux-yocto style" recipe.

Every linux-yocto style recipe must define the `KMACHINE` variable. This variable is typically set to the same value as the `MACHINE` variable, which is used by [BitBake](#). However, in some cases, the variable might instead refer to the underlying platform of the `MACHINE`.

Multiple BSPs can reuse the same `KMACHINE` name if they are built using the same BSP description. Multiple Corei7-based BSPs could share the same "intel-corei7-64" value for `KMACHINE`. It is important to realize that `KMACHINE` is just for kernel mapping, while `MACHINE` is the machine type within a BSP Layer. Even with this distinction, however, these two variables can hold the same value. See the [BSP Descriptions](#) section for more information.

Every linux-yocto style recipe must also indicate the Linux kernel source repository branch used to build the Linux kernel. The `KBRANCH` variable must be set to indicate the branch.

## Note

You can use the `KBRANCH` value to define an alternate branch typically with a machine override as shown here from the `meta-yocto-bsp` layer:

```
KBRANCH_edgerouter = "standard/edgerouter"
```

The linux-yocto style recipes can optionally define the following variables:

```
KERNEL_FEATURES
LINUX_KERNEL_TYPE
```

`LINUX_KERNEL_TYPE` defines the kernel type to be used in assembling the configuration. If you do not specify a `LINUX_KERNEL_TYPE`, it defaults to "standard". Together with `KMACHINE`, `LINUX_KERNEL_TYPE` defines the search arguments used by the kernel tools to find the appropriate description within the kernel Metadata with which to build out the sources and configuration. The linux-yocto recipes define "standard", "tiny", and "preempt-rt" kernel types. See the [Kernel Types](#) section for more information on kernel types.

During the build, the kern-tools search for the BSP description file that most closely matches the `KMACHINE` and `LINUX_KERNEL_TYPE` variables passed in from the recipe. The tools use the first BSP description it finds that match both variables. If the tools cannot find a match, they issue a warning.

The tools first search for the `KMACHINE` and then for the `LINUX_KERNEL_TYPE`. If the tools cannot find a partial match, they will use the sources from the `KBRANCH` and any configuration specified in the `SRC_URI`.

You can use the `KERNEL_FEATURES` variable to include features (configuration fragments, patches, or both) that are not already included by the `KMACHINE` and `LINUX_KERNEL_TYPE` variable combination. For example, to include a feature specified as "features/netfilter/netfilter.scc", specify:

```
KERNEL_FEATURES += "features/netfilter/netfilter.scc"
```

To include a feature called "cfg/sound.scc" just for the `qemux86` machine, specify:

```
KERNEL_FEATURES_append_qemux86 = " cfg/sound.scc"
```

The value of the entries in `KERNEL_FEATURES` are dependent on their location within the kernel Metadata itself. The examples here are taken from the `yocto-kernel-cache` repository. Each branch of this repository contains "features" and "cfg" subdirectories at the top-level. For more information, see the [Kernel Metadata Syntax](#) section.

## 16.3. Kernel Metadata Syntax¶

The kernel Metadata consists of three primary types of files: `SCC` <sup>[1]</sup> description files, configuration fragments, and patches. The `SCC` files define variables and include or otherwise reference any of the three file types. The description files are used to aggregate all types of kernel Metadata into what ultimately describes the sources and the configuration required to build a Linux kernel tailored to a specific machine.

The `SCC` description files are used to define two fundamental types of kernel Metadata:

- Features
- Board Support Packages (BSPs)

Features aggregate sources in the form of patches and configuration fragments into a modular reusable unit. You can use features to implement conceptually separate kernel Metadata descriptions such as pure configuration fragments, simple patches, complex features, and kernel types. [Kernel types](#) define general kernel features and policy to be reused in the BSPs.

BSPs define hardware-specific features and aggregate them with kernel types to form the final description of what will be assembled and built.

While the kernel Metadata syntax does not enforce any logical separation of configuration fragments, patches, features or kernel types, best practices dictate a logical separation of these types of Metadata. The following Metadata file hierarchy is recommended:

```
base/
 bsp/
 cfg/
 features/
 ktypes/
```

patches/

The `bsp` directory contains the [BSP descriptions](#). The remaining directories all contain "features". Separating `bsp` from the rest of the structure aids conceptualizing intended usage.

Use these guidelines to help place your `SCC` description files within the structure:

- If your file contains only configuration fragments, place the file in the `cfg` directory.
- If your file contains only source-code fixes, place the file in the `patches` directory.
- If your file encapsulates a major feature, often combining sources and configurations, place the file in `features` directory.
- If your file aggregates non-hardware configuration and patches in order to define a base kernel policy or major kernel type to be reused across multiple BSPs, place the file in `ktypes` directory.

These distinctions can easily become blurred - especially as out-of-tree features slowly merge upstream over time. Also, remember that how the description files are placed is a purely logical organization and has no impact on the functionality of the kernel Metadata. There is no impact because all of `cfg`, `features`, `patches`, and `ktypes`, contain "features" as far as the kernel tools are concerned.

Paths used in kernel Metadata files are relative to `base`, which is either [FILESEXTRAPATHS](#) if you are creating Metadata in [recipe-space](#), or the top level of [yocto-kernel-cache](#) if you are creating [Metadata outside of the recipe-space](#).

### 16.3.1. Configuration¶

The simplest unit of kernel Metadata is the configuration-only feature. This feature consists of one or more Linux kernel configuration parameters in a configuration fragment file (`.cfg`) and a `.scc` file that describes the fragment.

As an example, consider the Symmetric Multi-Processing (SMP) fragment used with the `linux-yocto-4.12` kernel as defined outside of the recipe space (i.e. `yocto-kernel-cache`). This Metadata consists of two files: `smp.scc` and `smp.cfg`. You can find these files in the `cfg` directory of the `yocto-4.12` branch in the `yocto-kernel-cache` Git repository:

```
cfg/smp.scc:
define KFEATURE_DESCRIPTION "Enable SMP for 32 bit builds"
define KFEATURE_COMPATIBILITY all

kconf hardware smp.cfg

cfg/smp.cfg:
CONFIG_SMP=y
CONFIG_SCHED_SMT=y
Increase default NR_CPUS from 8 to 64 so that platform with
more than 8 processors can be all activated at boot time
CONFIG_NR_CPUS=64
The following is needed when setting NR_CPUS to something
greater than 8 on x86 architectures, it should be automatically
disregarded by Kconfig when using a different arch
CONFIG_X86_BIGSMP=y
```

You can find general information on configuration fragment files in the ["Creating Configuration Fragments"](#) section.

Within the `smp.scc` file, the [KFEATURE\\_DESCRIPTION](#) statement provides a short description of the fragment. Higher level kernel tools use this description.

Also within the `smp.scc` file, the `kconf` command includes the actual configuration fragment in an `.scc` file, and the "hardware" keyword identifies the fragment as being hardware enabling, as opposed to general policy, which would use the "non-hardware" keyword. The distinction is made for the benefit of the configuration validation tools, which warn you if a hardware fragment overrides a policy set by a non-hardware fragment.

### Note

The description file can include multiple `kconf` statements, one per fragment.

As described in the ["Validating Configuration"](#) section, you can use the following BitBake command to audit your configuration:

```
$ bitbake linux-yocto -c kernel_configcheck -f
```

### 16.3.2. Patches¶

Patch descriptions are very similar to configuration fragment descriptions, which are described in the previous section. However, instead of a `.cfg` file, these descriptions work with source patches (i.e. `.patch` files).

A typical patch includes a description file and the patch itself. As an example, consider the build patches used with the `linux-yocto-4.12` kernel as defined outside of the recipe space (i.e. `yocto-kernel-cache`). This Metadata consists of several files: `build.scc` and a set of `*.patch` files. You can find these files in the `patches/build` directory of the `yocto-4.12` branch in the `yocto-kernel-cache` Git repository.

The following listings show the `build.scc` file and part of the `modpost-mask-trivial-warnings.patch` file:

```
patches/build/build.scc:
patch arm-serialize-build-targets.patch
patch powerpc-serialize-image-targets.patch
patch kbuild-exclude-meta-directory-from-distclean-processi.patch

applied by kgit
patch kbuild-add-meta-files-to-the-ignore-li.patch

patch modpost-mask-trivial-warnings.patch
```



```

patch menuconfig-check-lxdiaglog.sh-Allow-specification-of.patch

patches/build/modpost-mask-trivial-warnings.patch:
From bd48931bc142bdd104668f3a062a1f22600aae61 Mon Sep 17 00:00:00 2001
From: Paul Gortmaker <paul.gortmaker@windriver.com>
Date: Sun, 25 Jan 2009 17:58:09 -0500
Subject: [PATCH] modpost: mask trivial warnings

Newer HOSTCC will complain about various stdio fcns because
.
.
.
char *dump_write = NULL, *files_source = NULL;
int opt;
--
2.10.1

generated by cgit v0.10.2 at 2017-09-28 15:23:23 (GMT)

```

The description file can include multiple patch statements where each statement handles a single patch. In the example `build.scc` file, five patch statements exist for the five patches in the directory.

You can create a typical `.patch` file using `diff -Nurp` or `git format-patch` commands. For information on how to create patches, see the "[Using devtool to Patch the Kernel](#)" and "[Using Traditional Kernel Development to Patch the Kernel](#)" sections.

### 16.3.3. Features¶

Features are complex kernel Metadata types that consist of configuration fragments, patches, and possibly other feature description files. As an example, consider the following generic listing:

```

features/myfeature.scc
define KFEATURE_DESCRIPTION "Enable myfeature"

patch 0001-myfeature-core.patch
patch 0002-myfeature-interface.patch

include cfg/myfeature_dependency.scc
kconf non-hardware myfeature.cfg

```

This example shows how the `patch` and `kconf` commands are used as well as how an additional feature description file is included with the `include` command.

Typically, features are less granular than configuration fragments and are more likely than configuration fragments and patches to be the types of things you want to specify in the `KERNEL_FEATURES` variable of the Linux kernel recipe. See the "[Using Kernel Metadata in a Recipe](#)" section earlier in the manual.

### 16.3.4. Kernel Types¶

A kernel type defines a high-level kernel policy by aggregating non-hardware configuration fragments with patches you want to use when building a Linux kernel of a specific type (e.g. a real-time kernel). Syntactically, kernel types are no different than features as described in the "[Features](#)" section. The `LINUX_KERNEL_TYPE` variable in the kernel recipe selects the kernel type. For example, in the `linux-yocto_4.12.bb` kernel recipe found in `poky/meta/recipes-kernel/linux`, a `require` directive includes the `poky/meta/recipes-kernel/linux/linux-yocto.inc` file, which has the following statement that defines the default kernel type:

```
LINUX_KERNEL_TYPE ??= "standard"
```

Another example would be the real-time kernel (i.e. `linux-yocto-rt_4.12.bb`). This kernel recipe directly sets the kernel type as follows:

```
LINUX_KERNEL_TYPE = "preempt-rt"
```

#### Note

You can find kernel recipes in the `meta/recipes-kernel/linux` directory of the [Source Directory](#) (e.g. `poky/meta/recipes-kernel/linux/linux-yocto_4.12.bb`). See the "[Using Kernel Metadata in a Recipe](#)" section for more information.

Three kernel types ("standard", "tiny", and "preempt-rt") are supported for Linux Yocto kernels:

- "standard": Includes the generic Linux kernel policy of the Yocto Project linux-yocto kernel recipes. This policy includes, among other things, which file systems, networking options, core kernel features, and debugging and tracing options are supported.
- "preempt-rt": Applies the `PREEMPT_RT` patches and the configuration options required to build a real-time Linux kernel. This kernel type inherits from the "standard" kernel type.
- "tiny": Defines a bare minimum configuration meant to serve as a base for very small Linux kernels. The "tiny" kernel type is independent from the "standard" configuration. Although the "tiny" kernel type does not currently include any source changes, it might in the future.

For any given kernel type, the Metadata is defined by the `.scc` (e.g. `standard.scc`). Here is a partial listing for the `standard.scc` file, which is found in the `ktypes/standard` directory of the `yocto-kernel-cache` Git repository:

```

Include this kernel type fragment to get the standard features and
configuration values.

```



```
Note: if only the features are desired, but not the configuration
then this should be included as:
include ktypes/standard/standard.scc nocfg
if no chained configuration is desired, include it as:
include ktypes/standard/standard.scc nocfg inherit
```

```
include ktypes/base/base.scc
branch standard

kconf non-hardware standard.cfg

include features/kgdb/kgdb.scc
.
.
.

include cfg/net/ip6_nf.scc
include cfg/net/bridge.scc

include cfg/systemd.scc

include features/rfkill/rfkill.scc
```

As with any `.SCC` file, a kernel type definition can aggregate other `.SCC` files with `include` commands. These definitions can also directly pull in configuration fragments and patches with the `kconf` and `patch` commands, respectively.

## Note

It is not strictly necessary to create a kernel type `.SCC` file. The Board Support Package (BSP) file can implicitly define the kernel type using a `define KTYPE myktype` line. See the "[BSP Descriptions](#)" section for more information.

### 16.3.5. BSP Descriptions¶

BSP descriptions (i.e. `*.SCC` files) combine kernel types with hardware-specific features. The hardware-specific Metadata is typically defined independently in the BSP layer, and then aggregated with each supported kernel type.

## Note

For BSPs supported by the Yocto Project, the BSP description files are located in the `bsp` directory of the [yocto-kernel-cache](#) repository organized under the "Yocto Linux Kernel" heading in the [Yocto Project Source Repositories](#).

This section overviews the BSP description structure, the aggregation concepts, and presents a detailed example using a BSP supported by the Yocto Project (i.e. BeagleBone Board). For complete information on BSP layer file hierarchy, see the Yocto Project Board Support Package (BSP) Developer's Guide.

#### 16.3.5.1. Overview¶

For simplicity, consider the following root BSP layer description files for the BeagleBone board. These files employ both a structure and naming convention for consistency. The naming convention for the file is as follows:

```
bsp_root_name-kernel_type.scc
```

Here are some example root layer BSP filenames for the BeagleBone Board BSP, which is supported by the Yocto Project:

```
beaglebone-standard.scc
beaglebone-preempt-rt.scc
```

Each file uses the root name (i.e. "beaglebone") BSP name followed by the kernel type.

Examine the `beaglebone-standard.scc` file:

```
define KMACHINE beaglebone
define KTYPE standard
define KARCH arm

include ktypes/standard/standard.scc
branch beaglebone

include beaglebone.scc

default policy for standard kernels
include features/latencytop/latencytop.scc
include features/profiling/profiling.scc
```

Every top-level BSP description file should define the `KMACHINE`, `KTYPE`, and `KARCH` variables. These variables allow the OpenEmbedded build system to identify the description as meeting the criteria set by the recipe being built. This example supports the "beaglebone" machine for the "standard" kernel and the "arm" architecture.

Be aware that a hard link between the `KTYPE` variable and a kernel type description file does not exist. Thus, if you do not have the kernel type defined in your kernel Metadata as it is here, you only need to ensure that the `LINUX_KERNEL_TYPE` variable in the kernel recipe and the `KTYPE` variable in the BSP description file match.

To separate your kernel policy from your hardware configuration, you include a kernel type (`ktype`), such as "standard". In the previous example, this is done using the following:

```
include ktypes/standard/standard.scc
```

This file aggregates all the configuration fragments, patches, and features that make up your standard kernel policy. See the "[Kernel Types](#)" section for more information.

To aggregate common configurations and features specific to the kernel for *mybsp*, use the following:

```
include mybsp.scc
```

You can see that in the BeagleBone example with the following:

```
include beaglebone.scc
```

For information on how to break a complete `.config` file into the various configuration fragments, see the "[Creating Configuration Fragments](#)" section.

Finally, if you have any configurations specific to the hardware that are not in a `*.scc` file, you can include them as follows:

```
kconf hardware mybsp-extra.cfg
```

The BeagleBone example does not include these types of configurations. However, the Malta 32-bit board does ("mti-malta32"). Here is the `mti-malta32-le-standard.scc` file:

```
define KMACHINE mti-malta32-le
define KMACHINE qemuMipsel
define KTYPE standard
define KARCH mips

include ktypes/standard/standard.scc
branch mti-malta32

include mti-malta32.scc
kconf hardware mti-malta32-le.cfg
```

### 16.3.5.2. Example¶

Many real-world examples are more complex. Like any other `.scc` file, BSP descriptions can aggregate features. Consider the Minnow BSP definition given the `linux-yocto-4.4` branch of the `yocto-kernel-cache` (i.e. `yocto-kernel-cache/bsp/minnow/minnow.scc`):

### Note

Although the Minnow Board BSP is unused, the Metadata remains and is being used here just as an example.

```
include cfg/x86.scc
include features/eg20t/eg20t.scc
include cfg/dmaengine.scc
include features/power/intel.scc
include cfg/efi.scc
include features/usb/ehci-hcd.scc
include features/usb/ohci-hcd.scc
include features/usb/usb-gadgets.scc
include features/usb/touchscreen-composite.scc
include cfg/timer/hpet.scc
include features/leds/leds.scc
include features/spi/spidev.scc
include features/i2c/i2cdev.scc
include features/mei/mei-txe.scc

Earlyprintk and port debug requires 8250
kconf hardware cfg/8250.cfg

kconf hardware minnow.cfg
kconf hardware minnow-dev.cfg
```

The `minnow.scc` description file includes a hardware configuration fragment (`minnow.cfg`) specific to the Minnow BSP as well as several more general configuration fragments and features enabling hardware found on the machine. This `minnow.scc` description file is then included in each of the three "minnow" description files for the supported kernel types (i.e. "standard", "preempt-rt", and "tiny"). Consider the "minnow" description for the "standard" kernel type (i.e. `minnow-standard.scc`):

```
define KMACHINE minnow
define KTYPE standard
define KARCH i386

include ktypes/standard

include minnow.scc

Extra minnow configs above the minimal defined in minnow.scc
include cfg/efi-ext.scc
include features/media/media-all.scc
include features/sound/snd_hda_intel.scc

The following should really be in standard.scc
USB live-image support
include cfg/usb-mass-storage.scc
include cfg/boot-live.scc
```

```
Basic profiling
include features/latencytop/latencytop.scc
include features/profiling/profiling.scc

Requested drivers that don't have an existing scc
kconf hardware minnow-drivers-extra.cfg
```

The `include` command midway through the file includes the `minnow.scc` description that defines all enabled hardware for the BSP that is common to all kernel types. Using this command significantly reduces duplication.

Now consider the "minnow" description for the "tiny" kernel type (i.e. `minnow-tiny.scc`):

```
define KMACHINE minnow
define KTYPE tiny
define KARCH i386

include ktypes/tiny

include minnow.scc
```

As you might expect, the "tiny" description includes quite a bit less. In fact, it includes only the minimal policy defined by the "tiny" kernel type and the hardware-specific configuration required for booting the machine along with the most basic functionality of the system as defined in the base "minnow" description file.

Notice again the three critical variables: `KMACHINE`, `KTYPE`, and `KARCH`. Of these variables, only `KTYPE` has changed to specify the "tiny" kernel type.

## 16.4. Kernel Metadata Location¶

Kernel Metadata always exists outside of the kernel tree either defined in a kernel recipe (recipe-space) or outside of the recipe. Where you choose to define the Metadata depends on what you want to do and how you intend to work. Regardless of where you define the kernel Metadata, the syntax used applies equally.

If you are unfamiliar with the Linux kernel and only wish to apply a configuration and possibly a couple of patches provided to you by others, the recipe-space method is recommended. This method is also a good approach if you are working with Linux kernel sources you do not control or if you just do not want to maintain a Linux kernel Git repository on your own. For partial information on how you can define kernel Metadata in the recipe-space, see the "[Modifying an Existing Recipe](#)" section.

Conversely, if you are actively developing a kernel and are already maintaining a Linux kernel Git repository of your own, you might find it more convenient to work with kernel Metadata kept outside the recipe-space. Working with Metadata in this area can make iterative development of the Linux kernel more efficient outside of the BitBake environment.

### 16.4.1. Recipe-Space Metadata¶

When stored in recipe-space, the kernel Metadata files reside in a directory hierarchy below `FILESEXTRAPATHS`. For a linux-yocto recipe or for a Linux kernel recipe derived by copying and modifying `oe-core/meta-skeleton/recipes-kernel/linux/linux-yocto-custom.bb` to a recipe in your layer, `FILESEXTRAPATHS` is typically set to `${THISDIR}/${PN}`. See the "[Modifying an Existing Recipe](#)" section for more information.

Here is an example that shows a trivial tree of kernel Metadata stored in recipe-space within a BSP layer:

```
meta-my_bsp_layer/
|-- recipes-kernel
| |-- linux
| | |-- linux-yocto
| | | |-- bsp-standard.scc
| | | |-- bsp.cfg
| | |-- standard.cfg
```

When the Metadata is stored in recipe-space, you must take steps to ensure BitBake has the necessary information to decide what files to fetch and when they need to be fetched again. It is only necessary to specify the `.scc` files on the `SRC_URI`. BitBake parses them and fetches any files referenced in the `.scc` files by the `include`, `patch`, or `kconf` commands. Because of this, it is necessary to bump the recipe `PR` value when changing the content of files not explicitly listed in the `SRC_URI`.

If the BSP description is in recipe space, you cannot simply list the `*.scc` in the `SRC_URI` statement. You need to use the following form from your kernel append file:

```
SRC_URI_append_myplatform = " \
 file://myplatform;type=kmeta;destsuffix=myplatform \
"
```

### 16.4.2. Metadata Outside the Recipe-Space¶

When stored outside of the recipe-space, the kernel Metadata files reside in a separate repository. The OpenEmbedded build system adds the Metadata to the build as a "type=kmeta" repository through the `SRC_URI` variable. As an example, consider the following `SRC_URI` statement from the `linux-yocto_4.12.bb` kernel recipe:

```
SRC_URI = "git://git.yoctoproject.org/linux-yocto-4.12.git;name=machine;branch=${KBRANCH}; \
 git://git.yoctoproject.org/yocto-kernel-cache;type=kmeta;name=meta;branch=yocto-4.12;destsuffix=${KMETA}"
```

`${KMETA}`, in this context, is simply used to name the directory into which the Git fetcher places the Metadata. This behavior is no different than any multi-repository `SRC_URI` statement used in a recipe (e.g. see the previous section).

You can keep kernel Metadata in a "kernel-cache", which is a directory containing configuration fragments. As with any Metadata kept outside the recipe-space, you simply need to use the `SRC_URI` statement with the "type=kmeta" attribute. Doing so makes the kernel Metadata available during the configuration phase.

If you modify the Metadata, you must not forget to update the `SRCREV` statements in the kernel's recipe. In particular, you need to update the `SRCREV_meta` variable to match the commit in the `KMETA` branch you wish to use. Changing the data in these branches and not updating the `SRCREV` statements to match will cause the build to fetch an older commit.

## 16.5. Organizing Your Source¶

Many recipes based on the `linux-yocto-custom.bb` recipe use Linux kernel sources that have only a single branch - "master". This type of repository structure is fine for linear development supporting a single machine and architecture. However, if you work with multiple boards and architectures, a kernel source repository with multiple branches is more efficient. For example, suppose you need a series of patches for one board to boot. Sometimes, these patches are works-in-progress or fundamentally wrong, yet they are still necessary for specific boards. In these situations, you most likely do not want to include these patches in every kernel you build (i.e. have the patches as part of the lone "master" branch). It is situations like these that give rise to multiple branches used within a Linux kernel sources Git repository.

Repository organization strategies exist that maximize source reuse, remove redundancy, and logically order your changes. This section presents strategies for the following cases:

- Encapsulating patches in a feature description and only including the patches in the BSP descriptions of the applicable boards.
- Creating a machine branch in your kernel source repository and applying the patches on that branch only.
- Creating a feature branch in your kernel source repository and merging that branch into your BSP when needed.

The approach you take is entirely up to you and depends on what works best for your development model.

### 16.5.1. Encapsulating Patches¶

if you are reusing patches from an external tree and are not working on the patches, you might find the encapsulated feature to be appropriate. Given this scenario, you do not need to create any branches in the source repository. Rather, you just take the static patches you need and encapsulate them within a feature description. Once you have the feature description, you simply include that into the BSP description as described in the "[BSP Descriptions](#)" section.

You can find information on how to create patches and BSP descriptions in the "[Patches](#)" and "[BSP Descriptions](#)" sections.

### 16.5.2. Machine Branches¶

When you have multiple machines and architectures to support, or you are actively working on board support, it is more efficient to create branches in the repository based on individual machines. Having machine branches allows common source to remain in the "master" branch with any features specific to a machine stored in the appropriate machine branch. This organization method frees you from continually reintegrating your patches into a feature.

Once you have a new branch, you can set up your kernel Metadata to use the branch a couple different ways. In the recipe, you can specify the new branch as the `KBRANCH` to use for the board as follows:

```
KBRANCH = "mynewbranch"
```

Another method is to use the `branch` command in the BSP description:

```
mybsp.scc:
define KMACHINE mybsp
define KTYPE standard
define KARCH i386
include standard.scc

branch mynewbranch

include mybsp-hw.scc
```

If you find yourself with numerous branches, you might consider using a hierarchical branching system similar to what the Yocto Linux Kernel Git repositories use:

```
common/kernel_type/machine
```

If you had two kernel types, "standard" and "small" for instance, three machines, and `common` as `mydir`, the branches in your Git repository might look like this:

```
mydir/base
mydir/standard/base
mydir/standard/machine_a
mydir/standard/machine_b
mydir/standard/machine_c
mydir/small/base
mydir/small/machine_a
```

This organization can help clarify the branch relationships. In this case, `mydir/standard/machine_a` includes everything in `mydir/base` and `mydir/standard/base`. The "standard" and "small" branches add sources specific to those kernel types that for whatever reason are not appropriate for the other branches.

## Note

The "base" branches are an artifact of the way Git manages its data internally on the filesystem: Git will not allow you to use `mydir/standard` and `mydir/standard/machine_a` because it would have to create a file and a directory named "standard".

### 16.5.3. Feature Branches¶

When you are actively developing new features, it can be more efficient to work with that feature as a branch, rather than as a set of patches that have to be regularly updated. The Yocto Project Linux kernel tools provide for this with the `git merge` command.

To merge a feature branch into a BSP, insert the `git merge` command after any `branch` commands:

```
mybsp.scc:
 define KMACHINE mybsp
 define KTYPE standard
 define KARCH i386
 include standard.scc

 branch mynewbranch
 git merge myfeature

 include mybsp-hw.scc
```

## 16.6. SCC Description File Reference¶

This section provides a brief reference for the commands you can use within an SCC description file (`.scc`):

- `branch [ref]`: Creates a new branch relative to the current branch (typically `${KTYPE}`) using the currently checked-out branch, or "ref" if specified.
- `define`: Defines variables, such as `KMACHINE`, `KTYPE`, `KARCH`, and `KFEATURE_DESCRIPTION`.
- `include SCC_FILE`: Includes an SCC file in the current file. The file is parsed as if you had inserted it inline.
- `kconf [hardware|non-hardware] CFG_FILE`: Queues a configuration fragment for merging into the final Linux `.config` file.
- `git merge GIT_BRANCH`: Merges the feature branch into the current branch.
- `patch PATCH_FILE`: Applies the patch to the current Git branch.

[1] `scc` stands for Series Configuration Control, but the naming has less significance in the current implementation of the tooling than it had in the past. Consider `scc` files to be description files.

## Appendix D. Advanced Kernel Concepts¶

[D.1. Yocto Project Kernel Development and Maintenance](#)

[D.2. Yocto Linux Kernel Architecture and Branching Strategies](#)

[D.3. Kernel Build File Hierarchy](#)

[D.4. Determining Hardware and Non-Hardware Features for the Kernel Configuration Audit Phase](#)

### D.1. Yocto Project Kernel Development and Maintenance¶

Kernels available through the Yocto Project (Yocto Linux kernels), like other kernels, are based off the Linux kernel releases from <http://www.kernel.org>. At the beginning of a major Linux kernel development cycle, the Yocto Project team chooses a Linux kernel based on factors such as release timing, the anticipated release timing of final upstream `kernel.org` versions, and Yocto Project feature requirements. Typically, the Linux kernel chosen is in the final stages of development by the Linux community. In other words, the Linux kernel is in the release candidate or "rc" phase and has yet to reach final release. But, by being in the final stages of external development, the team knows that the `kernel.org` final release will clearly be within the early stages of the Yocto Project development window.

This balance allows the Yocto Project team to deliver the most up-to-date Yocto Linux kernel possible, while still ensuring that the team has a stable official release for the baseline Linux kernel version.

As implied earlier, the ultimate source for Yocto Linux kernels are released kernels from `kernel.org`. In addition to a foundational kernel from `kernel.org`, the available Yocto Linux kernels contain a mix of important new mainline developments, non-mainline developments (when no alternative exists), Board Support Package (BSP) developments, and custom features. These additions result in a commercially released Yocto Project Linux kernel that caters to specific embedded designer needs for targeted hardware.

You can find a web interface to the Yocto Linux kernels in the [Source Repositories](#) at <http://git.yoctoproject.org>. If you look at the interface, you will see to the left a grouping of Git repositories titled "Yocto Linux Kernel". Within this group, you will find several Linux Yocto kernels developed and included with Yocto Project releases:

- **linux-yocto-4.1**: The stable Yocto Project kernel to use with the Yocto Project Release 2.0. This kernel is based on the Linux 4.1 released kernel.
- **linux-yocto-4.4**: The stable Yocto Project kernel to use with the Yocto Project Release 2.1. This kernel is based on the Linux 4.4 released kernel.
- **linux-yocto-4.6**: A temporary kernel that is not tied to any Yocto Project release.
- **linux-yocto-4.8**: The stable Yocto Project kernel to use with the Yocto Project Release 2.2.
- **linux-yocto-4.9**: The stable Yocto Project kernel to use with the Yocto Project Release 2.3. This kernel is based on the Linux 4.9 released kernel.
- **linux-yocto-4.10**: The default stable Yocto Project kernel to use with the Yocto Project Release 2.3. This kernel is based on the Linux 4.10 released kernel.

- **linux-yocto-4.12:** The default stable Yocto Project kernel to use with the Yocto Project Release 2.4. This kernel is based on the Linux 4.12 released kernel.
- **yocto-kernel-cache:** The `linux-yocto-cache` contains patches and configurations for the linux-yocto kernel tree. This repository is useful when working on the linux-yocto kernel. For more information on this "Advanced Kernel Metadata", see the "Working With Advanced Metadata (`yocto-kernel-cache`)" Chapter.
- **linux-yocto-dev:** A development kernel based on the latest upstream release candidate available.

## Notes

Long Term Support Initiative (LTSI) for Yocto Linux kernels is as follows:

- For Yocto Project releases 1.7, 1.8, and 2.0, the LTSI kernel is `linux-yocto-3.14`.
- For Yocto Project releases 2.1, 2.2, and 2.3, the LTSI kernel is `linux-yocto-4.1`.
- For Yocto Project release 2.4, the LTSI kernel is `linux-yocto-4.9`
- `linux-yocto-4.4` is an LTS kernel.

Once a Yocto Linux kernel is officially released, the Yocto Project team goes into their next development cycle, or upward revision (uprev) cycle, while still continuing maintenance on the released kernel. It is important to note that the most sustainable and stable way to include feature development upstream is through a kernel uprev process. Back-porting hundreds of individual fixes and minor features from various kernel versions is not sustainable and can easily compromise quality.

During the uprev cycle, the Yocto Project team uses an ongoing analysis of Linux kernel development, BSP support, and release timing to select the best possible `kernel.org` Linux kernel version on which to base subsequent Yocto Linux kernel development. The team continually monitors Linux community kernel development to look for significant features of interest. The team does consider back-porting large features if they have a significant advantage. User or community demand can also trigger a back-port or creation of new functionality in the Yocto Project baseline kernel during the uprev cycle.

Generally speaking, every new Linux kernel both adds features and introduces new bugs. These consequences are the basic properties of upstream Linux kernel development and are managed by the Yocto Project team's Yocto Linux kernel development strategy. It is the Yocto Project team's policy to not back-port minor features to the released Yocto Linux kernel. They only consider back-porting significant technological jumps - and, that is done after a complete gap analysis. The reason for this policy is that back-porting any small to medium sized change from an evolving Linux kernel can easily create mismatches, incompatibilities and very subtle errors.

The policies described in this section result in both a stable and a cutting edge Yocto Linux kernel that mixes forward ports of existing Linux kernel features and significant and critical new functionality. Forward porting Linux kernel functionality into the Yocto Linux kernels available through the Yocto Project can be thought of as a "micro uprev." The many "micro uprevs" produce a Yocto Linux kernel version with a mix of important new mainline, non-mainline, BSP developments and feature integrations. This Yocto Linux kernel gives insight into new features and allows focused amounts of testing to be done on the kernel, which prevents surprises when selecting the next major uprev. The quality of these cutting edge Yocto Linux kernels is evolving and the kernels are used in leading edge feature and BSP development.

## D.2. Yocto Linux Kernel Architecture and Branching Strategies¶

As mentioned earlier, a key goal of the Yocto Project is to present the developer with a kernel that has a clear and continuous history that is visible to the user. The architecture and mechanisms, in particular the branching strategies, used achieve that goal in a manner similar to upstream Linux kernel development in `kernel.org`.

You can think of a Yocto Linux kernel as consisting of a baseline Linux kernel with added features logically structured on top of the baseline. The features are tagged and organized by way of a branching strategy implemented by the Yocto Project team using the Source Code Manager (SCM) Git.

## Notes

- Git is the obvious SCM for meeting the Yocto Linux kernel organizational and structural goals described in this section. Not only is Git the SCM for Linux kernel development in `kernel.org` but, Git continues to grow in popularity and supports many different work flows, front-ends and management techniques.
- You can find documentation on Git at <http://git-scm.com/documentation>. You can also get an introduction to Git as it applies to the Yocto Project in the "Git" section in the Yocto Project Overview and Concepts Manual. The latter reference provides an overview of Git and presents a minimal set of Git commands that allows you to be functional using Git. You can use as much, or as little, of what Git has to offer to accomplish what you need for your project. You do not have to be a "Git Expert" in order to use it with the Yocto Project.

Using Git's tagging and branching features, the Yocto Project team creates kernel branches at points where functionality is no longer shared and thus, needs to be isolated. For example, board-specific incompatibilities would require different functionality and would require a branch to separate the features. Likewise, for specific kernel features, the same branching strategy is used.

This "tree-like" architecture results in a structure that has features organized to be specific for particular functionality, single kernel types, or a subset of kernel types. Thus, the user has the ability to see the added features and the commits that make up those features. In addition to being able to see added features, the user can also view the history of what made up the baseline Linux kernel.

Another consequence of this strategy results in not having to store the same feature twice internally in the tree. Rather, the kernel team stores the unique differences required to apply the feature onto the kernel type in question.

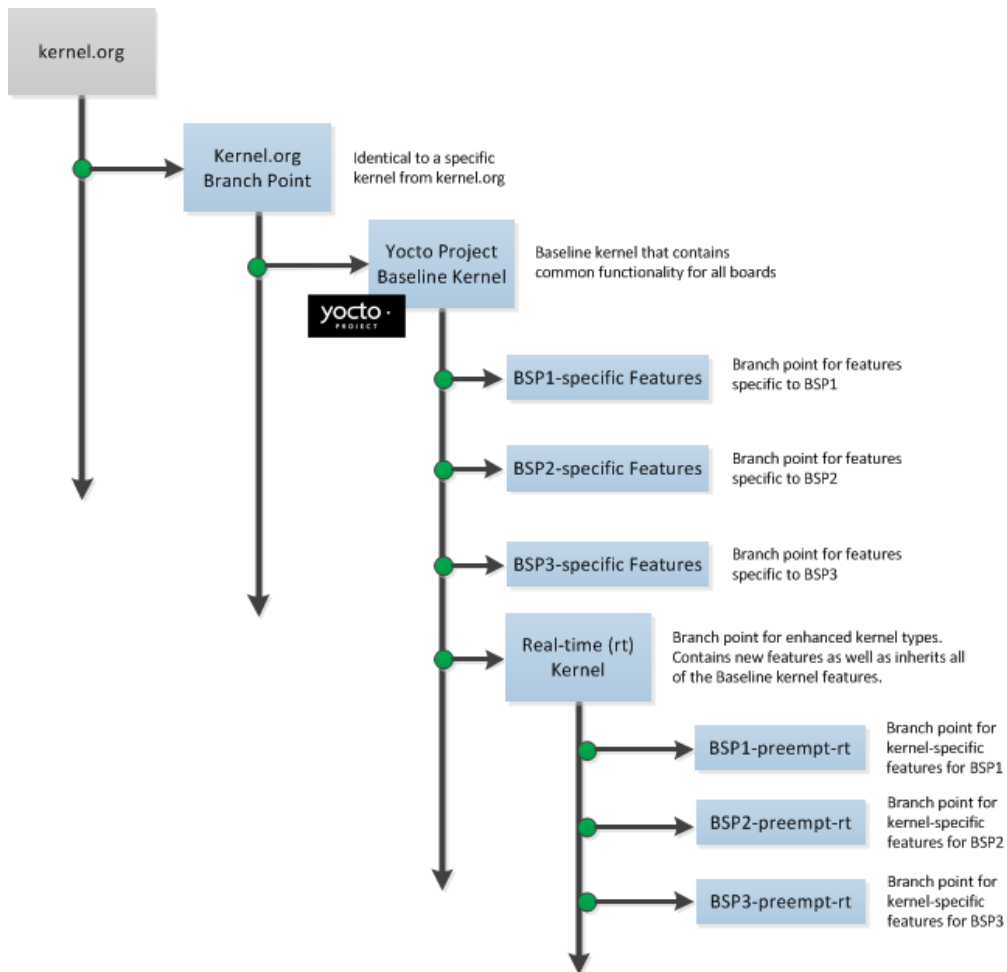
## Note

The Yocto Project team strives to place features in the tree such that features can be shared by all boards and kernel types where possible. However, during development cycles or when large features are merged, the team cannot always follow this practice. In those cases, the team uses isolated branches to merge features.

BSP-specific code additions are handled in a similar manner to kernel-specific additions. Some BSPs only make sense given certain kernel types. So, for these types, the team creates branches off the end of that kernel type for all of the BSPs that are supported on that kernel type. From the perspective of the tools that create the BSP branch, the BSP is really no different than a feature. Consequently, the same branching strategy applies to BSPs as it does to kernel features. So again, rather than store the BSP twice, the team only stores the unique differences for the BSP across the supported multiple kernels.

While this strategy can result in a tree with a significant number of branches, it is important to realize that from the developer's point of view, there is a linear path that travels from the baseline `kernel.org`, through a select group of features and ends with their BSP-specific commits. In other words, the divisions of the kernel are transparent and are not relevant to the developer on a day-to-day basis. From the developer's perspective, this path is the "master" branch in Git terms. The developer does not need to be aware of the existence of any other branches at all. Of course, value exists in the having these branches in the tree, should a person decide to explore them. For example, a comparison between two BSPs at either the commit level or at the line-by-line code `diff` level is now a trivial operation.

The following illustration shows the conceptual Yocto Linux kernel.



In the illustration, the "Kernel.org Branch Point" marks the specific spot (or Linux kernel release) from which the Yocto Linux kernel is created. From this point forward in the tree, features and differences are organized and tagged.

The "Yocto Project Baseline Kernel" contains functionality that is common to every kernel type and BSP that is organized further along in the tree. Placing these common features in the tree this way means features do not have to be duplicated along individual branches of the tree structure.

From the "Yocto Project Baseline Kernel", branch points represent specific functionality for individual Board Support Packages (BSPs) as well as real-time kernels. The illustration represents this through three BSP-specific branches and a real-time kernel branch. Each branch represents some unique functionality for the BSP or for a real-time Yocto Linux kernel.

In this example structure, the "Real-time (rt) Kernel" branch has common features for all real-time Yocto Linux kernels and contains more branches for individual BSP-specific kernels. The illustration shows three branches as an example. Each branch points the way to specific, unique features for a respective real-time kernel as they apply to a given BSP.

The resulting tree structure presents a clear path of markers (or branches) to the developer that, for all practical purposes, is the Yocto Linux kernel needed for any given set of requirements.

## Note

Keep in mind the figure does not take into account all the supported Yocto Linux kernels, but rather shows a single generic kernel just for conceptual purposes. Also keep in mind that this structure represents the Yocto

Project [Source Repositories](#) that are either pulled from during the build or established on the host development system prior to the build by either cloning a particular kernel's Git repository or by downloading and unpacking a tarball.

Working with the kernel as a structured tree follows recognized community best practices. In particular, the kernel as shipped with the product, should be considered an "upstream source" and viewed as a series of historical and documented modifications (commits). These modifications represent the development and stabilization done by the Yocto Project kernel development team.

Because commits only change at significant release points in the product life cycle, developers can work on a branch created from the last relevant commit in the shipped Yocto Project Linux kernel. As mentioned previously, the structure is transparent to the developer because the kernel tree is left in this state after cloning and building the kernel.

### D.3. Kernel Build File Hierarchy¶

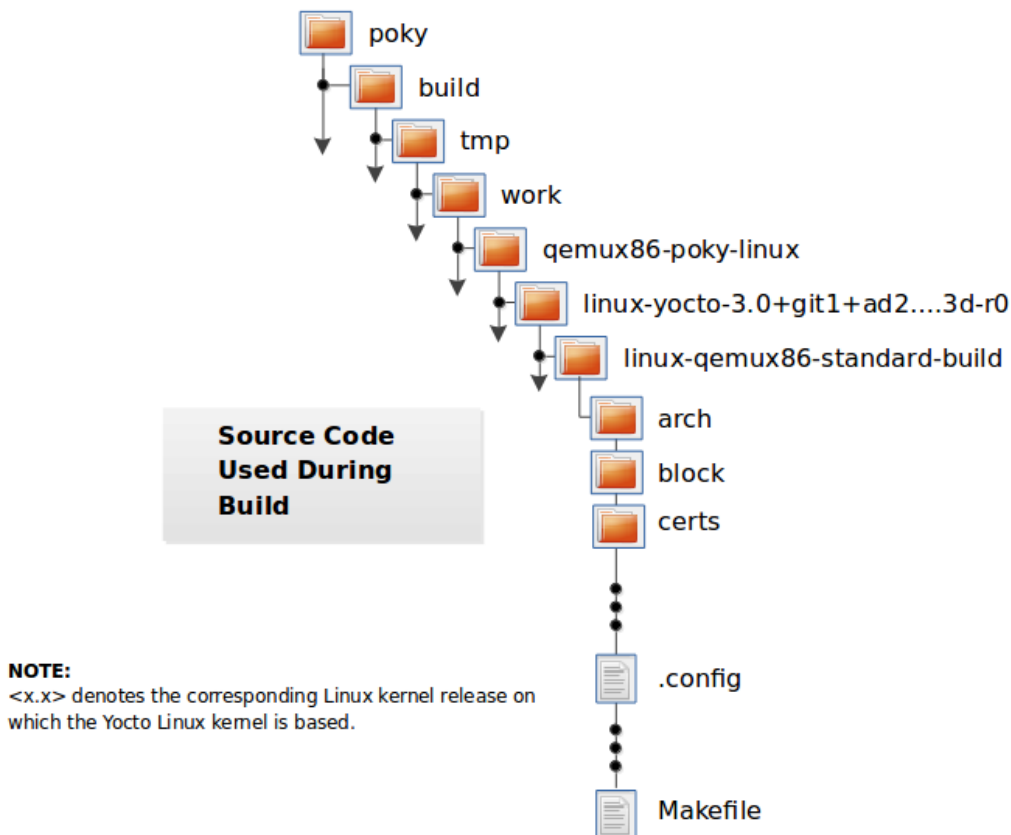
Upstream storage of all the available kernel source code is one thing, while representing and using the code on your host development system is another. Conceptually, you can think of the kernel source repositories as all the source files necessary for all the supported Yocto Linux kernels. As a developer, you are just interested in the source files for the kernel on which you are working. And, furthermore, you need them available on your host system.

Kernel source code is available on your host system several different ways:

- **Files Accessed While using `devtool`:** `devtool`, which is available with the Yocto Project, is the preferred method by which to modify the kernel. See the "[Kernel Modification Workflow](#)" section.
- **Cloned Repository:** If you are working in the kernel all the time, you probably would want to set up your own local Git repository of the Yocto Linux kernel tree. For information on how to clone a Yocto Linux kernel Git repository, see the "[Preparing the Build Host to Work on the Kernel](#)" section.
- **Temporary Source Files from a Build:** If you just need to make some patches to the kernel using a traditional BitBake workflow (i.e. not using the `devtool`), you can access temporary kernel source files that were extracted and used during a kernel build.

The temporary kernel source files resulting from a build using BitBake have a particular hierarchy. When you build the kernel on your development system, all files needed for the build are taken from the source repositories pointed to by the `SRC_URI` variable and gathered in a temporary work area where they are subsequently used to create the unique kernel. Thus, in a sense, the process constructs a local source tree specific to your kernel from which to generate the new kernel image.

The following figure shows the temporary file structure created on your host system when you build the kernel using Bitbake. This [Build Directory](#) contains all the source files used during the build.



Again, for additional information on the Yocto Project kernel's architecture and its branching strategy, see the "[Yocto Linux Kernel Architecture and Branching Strategies](#)" section. You can also reference the "[Using `devtool` to Patch the Kernel](#)" and "[Using Traditional Kernel Development to Patch the Kernel](#)" sections for detailed example that modifies the kernel.

### D.4. Determining Hardware and Non-Hardware Features for the Kernel Configuration Audit Phase¶



This section describes part of the kernel configuration audit phase that most developers can ignore. For general information on kernel configuration including `menuconfig`, `defconfig` files, and configuration fragments, see the "[Configuring the Kernel](#)" section.

During this part of the audit phase, the contents of the final `.config` file are compared against the fragments specified by the system. These fragments can be system fragments, distro fragments, or user-specified configuration elements. Regardless of their origin, the OpenEmbedded build system warns the user if a specific option is not included in the final kernel configuration.

By default, in order to not overwhelm the user with configuration warnings, the system only reports missing "hardware" options as they could result in a boot failure or indicate that important hardware is not available.

To determine whether or not a given option is "hardware" or "non-hardware", the kernel Metadata in `yocto-kernel-cache` contains files that classify individual or groups of options as either hardware or non-hardware. To better show this, consider a situation where the `yocto-kernel-cache` contains the following files:

```
yocto-kernel-cache/features/drm-psb/hardware.cfg
yocto-kernel-cache/features/kgdb/hardware.cfg
yocto-kernel-cache/ktypes/base/hardware.cfg
yocto-kernel-cache/bsp/mti-malta32/hardware.cfg
yocto-kernel-cache/bsp/fsl-mpc8315e-rdb/hardware.cfg
yocto-kernel-cache/bsp/qemu-ppc32/hardware.cfg
yocto-kernel-cache/bsp/qemuarm9/hardware.cfg
yocto-kernel-cache/bsp/mti-malta64/hardware.cfg
yocto-kernel-cache/bsp/arm-versatile-926ejs/hardware.cfg
yocto-kernel-cache/bsp/common-pc/hardware.cfg
yocto-kernel-cache/bsp/common-pc-64/hardware.cfg
yocto-kernel-cache/features/rfkill/non-hardware.cfg
yocto-kernel-cache/ktypes/base/non-hardware.cfg
yocto-kernel-cache/features/aufs/non-hardware.kcf
yocto-kernel-cache/features/ocf/non-hardware.kcf
yocto-kernel-cache/ktypes/base/non-hardware.kcf
yocto-kernel-cache/ktypes/base/hardware.kcf
yocto-kernel-cache/bsp/qemu-ppc32/hardware.kcf
```

The following list provides explanations for the various files:

- `hardware.kcf`: Specifies a list of kernel Kconfig files that contain hardware options only.
- `non-hardware.kcf`: Specifies a list of kernel Kconfig files that contain non-hardware options only.
- `hardware.cfg`: Specifies a list of kernel `CONFIG_` options that are hardware, regardless of whether or not they are within a Kconfig file specified by a hardware or non-hardware Kconfig file (i.e. `hardware.kcf` or `non-hardware.kcf`).
- `non-hardware.cfg`: Specifies a list of kernel `CONFIG_` options that are not hardware, regardless of whether or not they are within a Kconfig file specified by a hardware or non-hardware Kconfig file (i.e. `hardware.kcf` or `non-hardware.kcf`).

Here is a specific example using the `kernel-cache/bsp/mti-malta32/hardware.cfg`:

```
CONFIG_SERIAL_8250
CONFIG_SERIAL_8250_CONSOLE
CONFIG_SERIAL_8250_NR_UARTS
CONFIG_SERIAL_8250_PCI
CONFIG_SERIAL_CORE
CONFIG_SERIAL_CORE_CONSOLE
CONFIG_VGA_ARB
```

The kernel configuration audit automatically detects these files (hence the names must be exactly the ones discussed here), and uses them as inputs when generating warnings about the final `.config` file.

A user-specified kernel Metadata repository, or recipe space feature, can use these same files to classify options that are found within its `.cfg` files as hardware or non-hardware, to prevent the OpenEmbedded build system from producing an error or warning when an option is not in the final `.config` file.

## Appendix E. Kernel Maintenance¶

### [E.1. Tree Construction](#)

### [E.2. Build Strategy](#)

## E.1. Tree Construction¶

This section describes construction of the Yocto Project kernel source repositories as accomplished by the Yocto Project team to create Yocto Linux kernel repositories. These kernel repositories are found under the heading "Yocto Linux Kernel" at <http://git.yoctoproject.org> and are shipped as part of a Yocto Project release. The team creates these repositories by compiling and executing the set of feature descriptions for every BSP and feature in the product. Those feature descriptions list all necessary patches, configurations, branches, tags, and feature divisions found in a Yocto Linux kernel. Thus, the Yocto Project Linux kernel repository (or tree) and accompanying Metadata in the `yocto-kernel-cache` are built.

The existence of these repositories allow you to access and clone a particular Yocto Project Linux kernel repository and use it to build images based on their configurations and features.

You can find the files used to describe all the valid features and BSPs in the Yocto Project Linux kernel in any clone of the Yocto Project Linux kernel source repository and `yocto-kernel-cache` Git trees. For example, the following commands clone the Yocto Project baseline Linux kernel that branches off `linux.org` version 4.12 and the `yocto-kernel-cache`, which contains stores of kernel Metadata:

```
$ git clone git://git.yoctoproject.org/linux-yocto-4.12
$ git clone git://git.yoctoproject.org/linux-kernel-cache
```

For more information on how to set up a local Git repository of the Yocto Project Linux kernel files, see the "[Preparing the Build Host to Work on the Kernel](#)" section.

Once you have cloned the kernel Git repository and the cache of Metadata on your local machine, you can discover the branches that are available in the repository using the following Git command:

```
$ git branch -a
```

Checking out a branch allows you to work with a particular Yocto Linux kernel. For example, the following commands check out the "standard/beagleboard" branch of the Yocto Linux kernel repository and the "yocto-4.12" branch of the `yocto-kernel-cache` repository:

```
$ cd ~/linux-yocto-4.12
$ git checkout -b my-kernel-4.12 remotes/origin/standard/beagleboard
$ cd ~/linux-kernel-cache
$ git checkout -b my-4.12-metadata remotes/origin/yocto-4.12
```

## Note

Branches in the `yocto-kernel-cache` repository correspond to Yocto Linux kernel versions (e.g. "yocto-4.12", "yocto-4.10", "yocto-4.9", and so forth).

Once you have checked out and switched to appropriate branches, you can see a snapshot of all the kernel source files used to build that particular Yocto Linux kernel for a particular board.

To see the features and configurations for a particular Yocto Linux kernel, you need to examine the `yocto-kernel-cache` Git repository. As mentioned, branches in the `yocto-kernel-cache` repository correspond to Yocto Linux kernel versions (e.g. `yocto-4.12`). Branches contain descriptions in the form of `.scc` and `.cfg` files.

You should realize, however, that browsing your local `yocto-kernel-cache` repository for feature descriptions and patches is not an effective way to determine what is in a particular kernel branch. Instead, you should use Git directly to discover the changes in a branch. Using Git is an efficient and flexible way to inspect changes to the kernel.

## Note

Ground up reconstruction of the complete kernel tree is an action only taken by the Yocto Project team during an active development cycle. When you create a clone of the kernel Git repository, you are simply making it efficiently available for building and development.

The following steps describe what happens when the Yocto Project Team constructs the Yocto Project kernel source Git repository (or tree) found at <http://git.yoctoproject.org> given the introduction of a new top-level kernel feature or BSP. The following actions effectively provide the Metadata and create the tree that includes the new feature, patch, or BSP:

1. **Pass Feature to the OpenEmbedded Build System:** A top-level kernel feature is passed to the kernel build subsystem. Normally, this feature is a BSP for a particular kernel type.
2. **Locate Feature:** The file that describes the top-level feature is located by searching these system directories:
  - The in-tree kernel-cache directories, which are located in the `yocto-kernel-cache` repository organized under the "Yocto Linux Kernel" heading in the [Yocto Project Source Repositories](#).
  - Areas pointed to by `SRC_URI` statements found in kernel recipes

For a typical build, the target of the search is a feature description in an `.scc` file whose name follows this format (e.g. `beaglebone-standard.scc` and `beaglebone-preempt-rt.scc`):

```
bsp_root_name-kernel_type.scc
```

3. **Expand Feature:** Once located, the feature description is either expanded into a simple script of actions, or into an existing equivalent script that is already part of the shipped kernel.
4. **Append Extra Features:** Extra features are appended to the top-level feature description. These features can come from the `KERNEL_FEATURES` variable in recipes.
5. **Locate, Expand, and Append Each Feature:** Each extra feature is located, expanded and appended to the script as described in step three.
6. **Execute the Script:** The script is executed to produce files `.scc` and `.cfg` files in appropriate directories of the `yocto-kernel-cache` repository. These files are descriptions of all the branches, tags, patches and configurations that need to be applied to the base Git repository to completely create the source (build) branch for the new BSP or feature.
7. **Clone Base Repository:** The base repository is cloned, and the actions listed in the `yocto-kernel-cache` directories are applied to the tree.
8. **Perform Cleanup:** The Git repositories are left with the desired branches checked out and any required branching, patching and tagging has been performed.

The kernel tree and cache are ready for developer consumption to be locally cloned, configured, and built into a Yocto Project kernel specific to some target hardware.

## Notes

- The generated `yocto-kernel-cache` repository adds to the kernel as shipped with the Yocto Project release. Any add-ons and configuration data are applied to the end of an existing branch. The full repository generation that is found in the official Yocto Project kernel repositories at <http://git.yoctoproject.org> is the combination of all supported boards and configurations.
- The technique the Yocto Project team uses is flexible and allows for seamless blending of an immutable history with additional patches specific to a deployment. Any additions to the kernel become an integrated part of the branches.
- The full kernel tree that you see on <http://git.yoctoproject.org> is generated through repeating the above steps for all valid BSPs. The end result is a branched, clean history tree that makes up the kernel for a given release. You can see the script (`kgit-scc`) responsible for this in the `yocto-kernel-tools` repository.
- The steps used to construct the full kernel tree are the same steps that BitBake uses when it builds a kernel image.

## E.2. Build Strategy¶

Once you have cloned a Yocto Linux kernel repository and the cache repository (`yocto-kernel-cache`) onto your development system, you can consider the compilation phase of kernel development, which is building a kernel image. Some prerequisites exist that are validated by the build process before compilation starts:

- The `SRC_URI` points to the kernel Git repository.
- A BSP build branch with Metadata exists in the `yocto-kernel-cache` repository. The branch is based on the Yocto Linux kernel version and has configurations and features grouped under the `yocto-kernel-cache/bsp` directory. For example, features and configurations for the BeagleBone Board assuming a `linux-yocto_4.12` kernel reside in the following area of the `yocto-kernel-cache` repository:

```
yocto-kernel-cache/bsp/beaglebone
```

### Note

In the previous example, the "yocto-4.12" branch is checked out in the `yocto-kernel-cache` repository.

The OpenEmbedded build system makes sure these conditions exist before attempting compilation. Other means, however, do exist, such as as bootstrapping a BSP.

Before building a kernel, the build process verifies the tree and configures the kernel by processing all of the configuration "fragments" specified by feature descriptions in the `.SCC` files. As the features are compiled, associated kernel configuration fragments are noted and recorded in the series of directories in their compilation order. The fragments are migrated, pre-processed and passed to the Linux Kernel Configuration subsystem (`lkc`) as raw input in the form of a `.config` file. The `lkc` uses its own internal dependency constraints to do the final processing of that information and generates the final `.config` file that is used during compilation.

Using the board's architecture and other relevant values from the board's template, kernel compilation is started and a kernel image is produced.

The other thing that you notice once you configure a kernel is that the build process generates a build tree that is separate from your kernel's local Git source repository tree. This build tree has a name that uses the following form, where `{MACHINE}` is the metadata name of the machine (BSP) and "kernel\_type" is one of the Yocto Project supported kernel types (e.g. "standard"):

```
linux-{MACHINE}-kernel_type-build
```

The existing support in the `kernel.org` tree achieves this default functionality.

This behavior means that all the generated files for a particular machine or BSP are now in the build tree directory. The files include the final `.config` file, all the `.o` files, the `.a` files, and so forth. Since each machine or BSP has its own separate [Build Directory](#) in its own separate branch of the Git repository, you can easily switch between different builds.



# Yocto Project Profiling and Tracing Manual

## Chapter 17. Yocto Project Profiling and Tracing Manual¶

- [17.1. Introduction](#)
- [17.2. General Setup](#)

### 17.1. Introduction¶

Yocto bundles a number of tracing and profiling tools - this 'HOWTO' describes their basic usage and shows by example how to make use of them to examine application and system behavior.

The tools presented are for the most part completely open-ended and have quite good and/or extensive documentation of their own which can be used to solve just about any problem you might come across in Linux. Each section that describes a particular tool has links to that tool's documentation and website.

The purpose of this 'HOWTO' is to present a set of common and generally useful tracing and profiling idioms along with their application (as appropriate) to each tool, in the context of a general-purpose 'drill-down' methodology that can be applied to solving a large number (90%?) of problems. For help with more advanced usages and problems, please see the documentation and/or websites listed for each tool.

The final section of this 'HOWTO' is a collection of real-world examples which we'll be continually adding to as we solve more problems using the tools - feel free to add your own examples to the list!

## 17.2. General Setup¶

Most of the tools are available only in 'sdk' images or in images built after adding 'tools-profile' to your local.conf. So, in order to be able to access all of the tools described here, please first build and boot an 'sdk' image e.g.

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

or alternatively by adding 'tools-profile' to the EXTRA\_IMAGE\_FEATURES line in your local.conf:

```
EXTRA_IMAGE_FEATURES = "debug-tweaks tools-profile"
```

If you use the 'tools-profile' method, you don't need to build an sdk image - the tracing and profiling tools will be included in non-sdk images as well e.g.:

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

### Note

By default, the Yocto build system strips symbols from the binaries it packages, which makes it difficult to use some of the tools.

You can prevent that by setting the `INHIBIT_PACKAGE_STRIP` variable to "1" in your `local.conf` when you build the image:

```
INHIBIT_PACKAGE_STRIP = "1"
```

The above setting will noticeably increase the size of your image.

If you've already built a stripped image, you can generate debug packages (xxx-dbg) which you can manually install as needed.

To generate debug info for packages, you can add dbg-pkgs to EXTRA\_IMAGE\_FEATURES in local.conf. For example:

```
EXTRA_IMAGE_FEATURES = "debug-tweaks tools-profile dbg-pkgs"
```

Additionally, in order to generate the right type of debuginfo, we also need to set `PACKAGE_DEBUG_SPLIT_STYLE` in the `local.conf` file:

```
PACKAGE_DEBUG_SPLIT_STYLE = 'debug-file-directory'
```

## Chapter 18. Overall Architecture of the Linux Tracing and Profiling Tools¶

### 18.1. Architecture of the Tracing and Profiling Tools

### 18.1. Architecture of the Tracing and Profiling Tools¶

It may seem surprising to see a section covering an 'overall architecture' for what seems to be a random collection of tracing tools that together make up the Linux tracing and profiling space. The fact is, however, that in recent years this seemingly disparate set of tools has started to converge on a 'core' set of underlying mechanisms:

- static tracepoints
- dynamic tracepoints
  - kprobes
  - uprobes
- the perf\_events subsystem
- debugfs

**Tying it Together:** Rather than enumerating here how each tool makes use of these common mechanisms, textboxes like this will make note of the specific usages in each tool as they come up in the course of the text.

## Chapter 19. Basic Usage (with examples) for each of the Yocto Tracing Tools¶

|                                                    |
|----------------------------------------------------|
| 19.1. perf                                         |
| 19.1.1. Setup                                      |
| 19.1.2. Basic Usage                                |
| 19.1.3. Documentation                              |
| 19.2. ftrace                                       |
| 19.2.1. Setup                                      |
| 19.2.2. Basic ftrace usage                         |
| 19.2.3. The 'trace events' Subsystem               |
| 19.2.4. trace-cmd/kernelshark                      |
| 19.2.5. Documentation                              |
| 19.3. systemtap                                    |
| 19.3.1. Setup                                      |
| 19.3.2. Running a Script on a Target               |
| 19.3.3. Documentation                              |
| 19.4. Sysprof                                      |
| 19.4.1. Setup                                      |
| 19.4.2. Basic Usage                                |
| 19.4.3. Documentation                              |
| 19.5. LTTng (Linux Trace Toolkit, next generation) |
| 19.5.1. Setup                                      |
| 19.5.2. Collecting and Viewing Traces              |
| 19.5.3. Documentation                              |
| 19.6. blktrace                                     |
| 19.6.1. Setup                                      |
| 19.6.2. Basic Usage                                |
| 19.6.3. Documentation                              |

This chapter presents basic usage examples for each of the tracing tools.

## 19.1. perf

The 'perf' tool is the profiling and tracing tool that comes bundled with the Linux kernel.

Don't let the fact that it's part of the kernel fool you into thinking that it's only for tracing and profiling the kernel - you can indeed use it to trace and profile just the kernel, but you can also use it to profile specific applications separately (with or without kernel context), and you can also use it to trace and profile the kernel and all applications on the system simultaneously to gain a system-wide view of what's going on.

In many ways, perf aims to be a superset of all the tracing and profiling tools available in Linux today, including all the other tools covered in this HOWTO. The past couple of years have seen perf subsume a lot of the functionality of those other tools and, at the same time, those other tools have removed large portions of their previous functionality and replaced it with calls to the equivalent functionality now implemented by the perf subsystem. Extrapolation suggests that at some point those other tools will simply become completely redundant and go away; until then, we'll cover those other tools in these pages and in many cases show how the same things can be accomplished in perf and the other tools when it seems useful to do so.

The coverage below details some of the most common ways you'll likely want to apply the tool; full documentation can be found either within the tool itself or in the man pages at [perf\(1\)](#).

### 19.1.1. Setup

For this section, we'll assume you've already performed the basic setup outlined in the General Setup section.

In particular, you'll get the most mileage out of perf if you profile an image built with the following in your `local.conf` file:

```
INHIBIT_PACKAGE_STRIP = "1"
```

perf runs on the target system for the most part. You can archive profile data and copy it to the host for analysis, but for the rest of this document we assume you've ssh'ed to the host and will be running the perf commands on the target.

### 19.1.2. Basic Usage

The perf tool is pretty much self-documenting. To remind yourself of the available commands, simply type 'perf', which will show you basic usage along with the available perf subcommands:

```
root@crownbay:~# perf

usage: perf [--version] [--help] COMMAND [ARGS]

The most commonly used perf commands are:
annotate Read perf.data (created by perf record) and display annotated code
archive Create archive with object files with build-ids found in perf.data file
bench General framework for benchmark suites
buildid-cache Manage build-id cache.
buildid-list List the buildids in a perf.data file
diff Read two perf.data files and display the differential profile
evlist List the event names in a perf.data file
inject Filter to augment the events stream with additional information
kmem Tool to trace/measure kernel memory(slab) properties
kvm Tool to trace/measure kvm guest os
list List all symbolic event types
lock Analyze lock events
probe Define new dynamic tracepoints
record Run a command and record its profile into perf.data
report Read perf.data (created by perf record) and display the profile
sched Tool to trace/measure scheduler properties (latencies)
script Read perf.data (created by perf record) and display trace output
stat Run a command and gather performance counter statistics
test Runs sanity tests.
timechart Tool to visualize total system behavior during a workload
top System profiling tool.
```

See 'perf help COMMAND' for more information on a specific command.

### 19.1.2.1. Using perf to do Basic Profiling¶

As a simple test case, we'll profile the 'wget' of a fairly large file, which is a minimally interesting case because it has both file and network I/O aspects, and at least in the case of standard Yocto images, it's implemented as part of busybox, so the methods we use to analyze it can be used in a very similar way to the whole host of supported busybox applets in Yocto.

```
root@crownbay:~# rm linux-2.6.19.2.tar.bz2; \
wget http://downloads.yoctoproject.org/mirror/sources/linux-2.6.19.2.tar.bz2
```

The quickest and easiest way to get some basic overall data about what's going on for a particular workload is to profile it using 'perf stat'. 'perf stat' basically profiles using a few default counters and displays the summed counts at the end of the run:

```
root@crownbay:~# perf stat wget http://downloads.yoctoproject.org/mirror/sources/linux-2.6.19.2.tar.bz2
Connecting to downloads.yoctoproject.org (140.211.169.59:80)
linux-2.6.19.2.tar.b 100% |*****| 41727k 0:00:00 ETA

Performance counter stats for 'wget http://downloads.yoctoproject.org/mirror/sources/linux-2.6.19.2.tar.bz2':

 4597.223902 task-clock # 0.077 CPUs utilized
 23568 context-switches # 0.005 M/sec
 68 CPU-migrations # 0.015 K/sec
 241 page-faults # 0.052 K/sec
 3045817293 cycles # 0.663 GHz
<not supported> stalled-cycles-frontend
<not supported> stalled-cycles-backend
 858909167 instructions # 0.28 insns per cycle
 165441165 branches # 35.987 M/sec
 19550329 branch-misses # 11.82% of all branches

 59.836627620 seconds time elapsed
```

Many times such a simple-minded test doesn't yield much of interest, but sometimes it does (see Real-world Yocto bug (slow loop-mounted write speed)).

Also, note that 'perf stat' isn't restricted to a fixed set of counters - basically any event listed in the output of 'perf list' can be tallied by 'perf stat'. For example, suppose we wanted to see a summary of all the events related to kernel memory allocation/freeing along with cache hits and misses:

```
root@crownbay:~# perf stat -e kmem:* -e cache-references -e cache-misses wget http://downloads.yoctoproject.org/mirror/so
Connecting to downloads.yoctoproject.org (140.211.169.59:80)
linux-2.6.19.2.tar.b 100% |*****| 41727k 0:00:00 ETA

Performance counter stats for 'wget http://downloads.yoctoproject.org/mirror/sources/linux-2.6.19.2.tar.bz2':

 5566 kmem:kmalloc
 125517 kmem:kmem_cache_alloc
 0 kmem:kmalloc_node
 0 kmem:kmem_cache_alloc_node
 34401 kmem:kfree
 69920 kmem:kmem_cache_free
 133 kmem:mm_page_free
 41 kmem:mm_page_free_batched
 11502 kmem:mm_page_alloc
 11375 kmem:mm_page_alloc_zone_locked
 0 kmem:mm_page_pcpu_drain
 0 kmem:mm_page_alloc_extfrag
 66848602 cache-references
 2917740 cache-misses # 4.365 % of all cache refs

 44.831023415 seconds time elapsed
```

So 'perf stat' gives us a nice easy way to get a quick overview of what might be happening for a set of events, but normally we'd need a little more detail in order to understand what's going on in a way that we can act on in a useful way.

To dive down into a next level of detail, we can use 'perf record'/'perf report' which will collect profiling data and present it to use using an interactive text-based UI (or simply as text if we specify --stdio to 'perf report').

As our first attempt at profiling this workload, we'll simply run 'perf record', handing it the workload we want to profile (everything after 'perf record' and any perf options we hand it - here none - will be executed in a new shell). perf collects samples until the process exits and records them in a file named 'perf.data' in the current working directory.

```
root@crownbay:~# perf record wget http://downloads.yoctoproject.org/mirror/sources/linux-2.6.19.2.tar.bz2

Connecting to downloads.yoctoproject.org (140.211.169.59:80)
linux-2.6.19.2.tar.b 100% |*****| 41727k 0:00:00 ETA
[perf record: Woken up 1 times to write data]
[perf record: Captured and wrote 0.176 MB perf.data (~7700 samples)]
```

To see the results in a 'text-based UI' (tui), simply run 'perf report', which will read the perf.data file in the current working directory and display the results in an interactive UI:

```
root@crownbay:~# perf report
```

```

trz@empanada:~
File Edit View Search Terminal Help
Events: 4K cycles, Thread: wget(1231)
3.86% wget [kernel.kallsyms] [k] sub_preempt_count
3.47% wget [kernel.kallsyms] [k] add_preempt_count
3.16% wget [kernel.kallsyms] [k] read_hpet
1.95% wget [kernel.kallsyms] [k] _copy_to_user_ll
1.92% wget [kernel.kallsyms] [k] system_call
1.82% wget [kernel.kallsyms] [k] _copy_from_user_ll_nozero
1.76% wget [kernel.kallsyms] [k] get_parent_ip
1.67% wget [kernel.kallsyms] [k] ext3_mark_inode_dirty
1.67% wget busybox [.] 0x000651e3
1.54% wget [kernel.kallsyms] [k] _find_get_block
1.37% wget [kernel.kallsyms] [k] ext3_get_blocks_handle
1.22% wget [kernel.kallsyms] [k] do_sys_poll
1.16% wget [kernel.kallsyms] [k] journal_add_journal_head
1.10% wget [kernel.kallsyms] [k] _block_write_begin
1.07% wget [kernel.kallsyms] [k] tcp_recvmmsg
1.02% wget [kernel.kallsyms] [k] in_lock_functions
1.01% wget [kernel.kallsyms] [k] ext3_new_blocks
0.97% wget [kernel.kallsyms] [k] do_get_write_access
0.94% wget [kernel.kallsyms] [k] memset
0.88% wget [kernel.kallsyms] [k] fget_light
0.84% wget [kernel.kallsyms] [k] ioread32
0.82% wget [kernel.kallsyms] [k] _ext3_get_inode_loc
0.79% wget [kernel.kallsyms] [k] bit_waitqueue
0.78% wget [kernel.kallsyms] [k] _schedule
0.74% wget libc-2.16.so [.] read
0.73% wget [kernel.kallsyms] [k] journal_dirty_metadata
0.73% wget [kernel.kallsyms] [k] _raw_spin_lock
0.72% wget libc-2.16.so [.] 0x0006f4de
0.67% wget [kernel.kallsyms] [k] fsnotify
0.67% wget [kernel.kallsyms] [k] kmem_cache_alloc
0.67% wget [kernel.kallsyms] [k] debug_smp_processor_id
0.66% wget [kernel.kallsyms] [k] ext3_ordered_write_end
0.64% wget [kernel.kallsyms] [k] journal_dirty_data
0.62% wget [kernel.kallsyms] [k] find_busiest_group
0.62% wget [kernel.kallsyms] [k] _rcu_read_unlock
0.61% wget [kernel.kallsyms] [k] radix_tree_lookup_element
0.60% wget [kernel.kallsyms] [k] kmem_cache_free
0.59% wget [kernel.kallsyms] [k] kfree
0.58% wget [kernel.kallsyms] [k] _mark_inode_dirty
0.57% wget [kernel.kallsyms] [k] ext3_journal_start_sb
Press '?' for help on key bindings

```

The above screenshot displays a 'flat' profile, one entry for each 'bucket' corresponding to the functions that were profiled during the profiling run, ordered from the most popular to the least (perf has options to sort in various orders and keys as well as display entries only above a certain threshold and so on - see the perf documentation for details). Note that this includes both userspace functions (entries containing a [.] ) and kernel functions accounted to the process (entries containing a [k]). (perf has command-line modifiers that can be used to restrict the profiling to kernel or userspace, among others).

Notice also that the above report shows an entry for 'busybox', which is the executable that implements 'wget' in Yocto, but that instead of a useful function name in that entry, it displays a not-so-friendly hex value instead. The steps below will show how to fix that problem.

Before we do that, however, let's try running a different profile, one which shows something a little more interesting. The only difference between the new profile and the previous one is that we'll add the -g option, which will record not just the address of a sampled function, but the entire callchain to the sampled function as well:

```

root@crownbay:~# perf record -g wget http://downloads.yoctoproject.org/mirror/sources/linux-2.6.19.2.tar.bz2
Connecting to downloads.yoctoproject.org (140.211.169.59:80)
linux-2.6.19.2.tar.b 100% | *****| 41727k 0:00:00 ETA
[perf record: Woken up 3 times to write data]
[perf record: Captured and wrote 0.652 MB perf.data (~28476 samples)]

```

```

root@crownbay:~# perf report

```

```

trz@empanada:~
File Edit View Search Terminal Help
Events: 4K cycles
+ 1.77% wget [kernel.kallsyms] [k] __copy_from_user_ll_nzero
- 1.77% wget [kernel.kallsyms] [k] __copy_to_user_ll
 - __copy_to_user_ll
 - 90.87% copy_to_user
 - 97.79% memcpy_toiovec
 - skb_copy_datagram_iovec
 - 89.99% tcp_recvmmsg
 inet_recvmmsg
 sock_aio_read
 do_sync_read
 vfs_read
 sys_read
 syscall_call
 read
 - 10.01% skb_copy_datagram_iovec
 tcp_recvmmsg
 inet_recvmmsg
 sock_aio_read
 do_sync_read
 vfs_read
 sys_read
 syscall_call
 read
 + 2.21% sys_clock_gettime
 - 6.75% sys_clock_gettime
 syscall_call
 syscall
 0x808c05
 - 2.38% memcpy_toiovec
 skb_copy_datagram_iovec
 tcp_recvmmsg
 inet_recvmmsg
 sock_aio_read
 do_sync_read
 vfs_read
 sys_read
 syscall_call
 read
+ 1.43% wget [kernel.kallsyms] [k] ext3_get_blocks_handle
+ 1.42% wget [kernel.kallsyms] [k] tcp_recvmmsg
Press '?' for help on key bindings

```

Using the callgraph view, we can actually see not only which functions took the most time, but we can also see a summary of how those functions were called and learn something about how the program interacts with the kernel in the process.

Notice that each entry in the above screenshot now contains a '+' on the left-hand side. This means that we can expand the entry and drill down into the callchains that feed into that entry. Pressing 'enter' on any one of them will expand the callchain (you can also press 'E' to expand them all at the same time or 'C' to collapse them all).

In the screenshot above, we've toggled the `__copy_to_user_ll()` entry and several subnodes all the way down. This lets us see which callchains contributed to the profiled `__copy_to_user_ll()` function which contributed 1.77% to the total profile.

As a bit of background explanation for these callchains, think about what happens at a high level when you run `wget` to get a file out on the network. Basically what happens is that the data comes into the kernel via the network connection (socket) and is passed to the userspace program '`wget`' (which is actually a part of `busybox`, but that's not important for now), which takes the buffers the kernel passes to it and writes it to a disk file to save it.

The part of this process that we're looking at in the above call stacks is the part where the kernel passes the data it's read from the socket down to `wget` i.e. a copy-to-user.

Notice also that here there's also a case where the hex value is displayed in the callstack, here in the expanded `sys_clock_gettime()` function. Later we'll see it resolve to a userspace function call in `busybox`.



```

trz@empanada:~
File Edit View Search Terminal Help
Events: 4K cycles
+ 3.60% wget [kernel.kallsyms] [k] sub_preempt_count
+ 3.12% wget [kernel.kallsyms] [k] read_hpet
+ 2.73% wget [kernel.kallsyms] [k] add_preempt_count
+ 2.48% wget busybox [.] 0x0000c0dd
+ 2.21% wget [kernel.kallsyms] [k] get_parent_ip
+ 1.94% wget [kernel.kallsyms] [k] system_call
+ 1.88% wget [kernel.kallsyms] [k] __find_get_block
- 1.77% wget [kernel.kallsyms] [k] __copy_from_user_ll_nzero
- 91.57% iov_iter_copy_from_user_atomic
 generic_file_buffered_write
 __generic_file_aio_write
 generic_file_aio_write
 do_sync_write
 vfs_write
 sys_write
 syscall_call
 write
- 8.43% generic_file_buffered_write
 __generic_file_aio_write
 generic_file_aio_write
 do_sync_write
 vfs_write
 sys_write
 syscall_call
 write
+ 1.77% wget [kernel.kallsyms] [k] __copy_to_user_ll
+ 1.43% wget [kernel.kallsyms] [k] ext3_get_blocks_handle
+ 1.42% wget [kernel.kallsyms] [k] tcp_recvmg
+ 1.41% wget [kernel.kallsyms] [k] ext3_new_blocks
+ 1.34% wget [kernel.kallsyms] [k] block_write_begin
+ 1.32% wget [kernel.kallsyms] [k] in_lock_functions
+ 1.32% wget [kernel.kallsyms] [k] ext3_mark_inode_dirty
+ 1.17% wget [kernel.kallsyms] [k] memset
+ 1.13% wget [kernel.kallsyms] [k] debug_smp_processor_id
+ 1.08% wget [kernel.kallsyms] [k] schedule
+ 1.05% wget [kernel.kallsyms] [k] journal_add_journal_head
+ 0.92% wget [kernel.kallsyms] [k] ext3_get_inode_loc
+ 0.83% wget [kernel.kallsyms] [k] do_sys_poll
+ 0.81% wget [kernel.kallsyms] [k] journal_dirty_metadata
Press '?' for help on key bindings

```

The above screenshot shows the other half of the journey for the data - from the wget program's userspace buffers to disk. To get the buffers to disk, the wget program issues a write(2), which does a copy-from-user to the kernel, which then takes care via some circuitous path (probably also present somewhere in the profile data), to get it safely to disk.

Now that we've seen the basic layout of the profile data and the basics of how to extract useful information out of it, let's get back to the task at hand and see if we can get some basic idea about where the time is spent in the program we're profiling, wget. Remember that wget is actually implemented as an applet in busybox, so while the process name is 'wget', the executable we're actually interested in is busybox. So let's expand the first entry containing busybox:

```

trz@empanada:~
File Edit View Search Terminal Help
Events: 4K cycles
+ 3.60% wget [kernel.kallsyms] [k] sub_preempt_count
+ 3.12% wget [kernel.kallsyms] [k] read_hpet
+ 2.73% wget [kernel.kallsyms] [k] add_preempt_count
- 2.48% wget busybox [.] 0x0000c0dd
 0x808c00
 0x80530d
 0x804bf1
 0x80868e
 0x80862e
 0x805310
 0x804b50
 0x80530a
 0x80ac1d
 0x80ac1b
 0x804bbb
 0x808c05
 0x8086aa
 0x808c05
 0x80833d
 0x808c05
 0x808315
 0x805307
 0x808c03
 0x80531a
 0x805311
 0x805307
 0x805309
 0x808318
 0x808c05
 0x808691
 0x80531b
 0x808c03
 0x808c02
 0x80868f
 0x80869b
 0x808699
 0x808c05
 0x805311
 0x808316
 0x808c05
Press '?' for help on key bindings

```

Again, before we expanded we saw that the function was labeled with a hex value instead of a symbol as with most of the kernel entries. Expanding the busybox entry doesn't make it any better.

The problem is that perf can't find the symbol information for the busybox binary, which is actually stripped out by the Yocto build system.

One way around that is to put the following in your `local.conf` file when you build the image:

```
INHIBIT_PACKAGE_STRIP = "1"
```

However, we already have an image with the binaries stripped, so what can we do to get perf to resolve the symbols? Basically we need to install the debuginfo for the busybox package.

To generate the debug info for the packages in the image, we can add `dbg-pkgs` to `EXTRA_IMAGE_FEATURES` in `local.conf`. For example:

```
EXTRA_IMAGE_FEATURES = "debug-tweaks tools-profile dbg-pkgs"
```

Additionally, in order to generate the type of debuginfo that perf understands, we also need to set `PACKAGE_DEBUG_SPLIT_STYLE` in the `local.conf` file:

```
PACKAGE_DEBUG_SPLIT_STYLE = 'debug-file-directory'
```

Once we've done that, we can install the debuginfo for busybox. The debug packages once built can be found in `build/tmp/deploy/rpm/*` on the host system. Find the `busybox-dbg-...rpm` file and copy it to the target. For example:

```
[trz@empanada core2]$ scp /home/trz/yocto/crownbay-tracing-dbg/build/tmp/deploy/rpm/core2_32/busybox-dbg-1.20.2-r2.core2_
root@192.168.1.31's password:
busybox-dbg-1.20.2-r2.core2_32.rpm 100% 1826KB 1.8MB/s 00:01
```

Now install the debug rpm on the target:

```
root@crownbay:~# rpm -i busybox-dbg-1.20.2-r2.core2_32.rpm
```

Now that the debuginfo is installed, we see that the busybox entries now display their functions symbolically:

```

File Edit View Search Terminal Help
Events: 4K cycles
+ 3.60% wget [kernel.kallsyms] [k] sub_preempt_count
+ 3.12% wget [kernel.kallsyms] [k] read_hpet
+ 2.73% wget [kernel.kallsyms] [k] add_preempt_count
+ 2.21% wget [kernel.kallsyms] [k] get_parent_ip
+ 1.94% wget [kernel.kallsyms] [k] system_call
+ 1.88% wget [kernel.kallsyms] [k] __find_get_block
+ 1.77% wget [kernel.kallsyms] [k] __copy_from_user_ll_nzero
+ 1.77% wget [kernel.kallsyms] [k] __copy_to_user_ll
+ 1.43% wget [kernel.kallsyms] [k] ext3_get_blocks_handle
+ 1.42% wget [kernel.kallsyms] [k] tcp_recvmg
+ 1.41% wget [kernel.kallsyms] [k] ext3_new_blocks
+ 1.34% wget [kernel.kallsyms] [k] __block_write_begin
+ 1.32% wget [kernel.kallsyms] [k] in_lock_functions
+ 1.32% wget [kernel.kallsyms] [k] ext3_mark_inode_dirty
+ 1.17% wget [kernel.kallsyms] [k] memset
+ 1.13% wget [kernel.kallsyms] [k] debug_smp_processor_id
+ 1.08% wget [kernel.kallsyms] [k] __schedule
+ 1.05% wget [kernel.kallsyms] [k] journal_add_journal_head
+ 0.92% wget [kernel.kallsyms] [k] __ext3_get_inode_loc
+ 0.83% wget [kernel.kallsyms] [k] do_sys_poll
+ 0.81% wget [kernel.kallsyms] [k] journal_dirty_metadata
+ 0.75% wget libc-2.16.so [.] 0x00115093
+ 0.74% wget [kernel.kallsyms] [k] find_busiest_group
+ 0.73% wget [kernel.kallsyms] [k] bit_waitqueue
+ 0.73% wget libc-2.16.so [.] read
- 0.67% wget busybox [.] udhcpc_main
udhcpc_main
+ 0.67% wget [kernel.kallsyms] [k] restore_nocheck
+ 0.66% wget [kernel.kallsyms] [k] radix_tree_lookup_element
+ 0.63% wget [kernel.kallsyms] [k] journal_stop
+ 0.63% wget [kernel.kallsyms] [k] kmem_cache_alloc
+ 0.62% wget [kernel.kallsyms] [k] tcp_poll
+ 0.62% wget [kernel.kallsyms] [k] do_get_write_access
+ 0.61% wget [kernel.kallsyms] [k] ext3_ordered_write_end
+ 0.61% wget libc-2.16.so [.] clearerr
+ 0.60% wget [kernel.kallsyms] [k] ext3_write_begin
+ 0.59% wget [kernel.kallsyms] [k] sched_clock_local
+ 0.58% wget [kernel.kallsyms] [k] fsnotify
+ 0.58% wget [kernel.kallsyms] [k] journal_put_journal_head
+ 0.55% wget [kernel.kallsyms] [k] __rcu_read_unlock
Press '?' for help on key bindings

```

If we expand one of the entries and press 'enter' on a leaf node, we're presented with a menu of actions we can take to get more information related to that entry:

```

trz@empanada:~
File Edit View Search Terminal Help
Annotate udhpcp_main
Zoom into wget(1241) thread
Zoom into busybox DS0
Browse map details
Exit
ESC: exit, ENTER|->: Select option

```

One of these actions allows us to show a view that displays a busybox-centric view of the profiled functions (in this case we've also expanded all the nodes using the 'E' key):

```

trz@empanada:~
File Edit View Search Terminal Help
Events: 103 cycles, DS0: busybox
- 27.21% wget [.] udhpcp_main
 udhpcp_main
- 14.67% wget [.] handle_input
 handle_input
- 8.95% wget [.] common_ping_main
 common_ping_main
 100.00% handle_input
- 8.94% wget [.] tftp_main
 tftp_main
 100.00% handle_input
- 8.30% wget [.] 0x000651d4
 0x80ac1d
 0x80ac1b
 0x80ac1e
 0x80ac1b
- 5.79% wget [.] INET_setroute
 INET_setroute
- 5.44% wget [.] doexit
 doexit
- 4.85% wget [.] bb_init_module
 bb_init_module
 100.00% handle_input
- 4.48% wget [.] handle_net_output
 handle_net_output
- 3.67% wget [.] ife_print
 ife_print
- 3.66% wget [.] load_modules_dep
 load_modules_dep
 handle_input
- 3.36% wget [.] gather_options_str
 gather_options_str
 100.00% handle_input
- 0.68% wget [.] nslookup_main
 nslookup_main

Press '?' for help on key bindings

```

Finally, we can see that now that the busybox debuginfo is installed, the previously unresolved symbol in the `sys_clock_gettime()` entry mentioned previously is now resolved, and shows that the `sys_clock_gettime` system call that was the source of 6.75% of the copy-to-user overhead was initiated by the `handle_input()` busybox function:

```

trz@empanada:~
File Edit View Search Terminal Help
Events: 4K cycles
+ 1.77% wget [kernel.kallsyms] [k] __copy_from_user_ll_nzero
- 1.77% wget [kernel.kallsyms] [k] __copy_to_user_ll
 - __copy_to_user_ll
 - 90.87% copy_to_user
 - 97.79% memcpy_toiovec
 - skb_copy_datagram_iovec
 - 89.99% tcp_recvmmsg
 inet_recvmmsg
 sock_aio_read
 do_sync_read
 vfs_read
 sys_read
 syscall_call
 read
 - 10.01% skb_copy_datagram_iovec
 tcp_recvmmsg
 inet_recvmmsg
 sock_aio_read
 do_sync_read
 vfs_read
 sys_read
 syscall_call
 read
 + 2.21% sys_clock_gettime
 - 6.75% sys_clock_gettime
 syscall_call
 syscall
 handle_input
 - 2.38% memcpy_toiovec
 skb_copy_datagram_iovec
 tcp_recvmmsg
 inet_recvmmsg
 sock_aio_read
 do_sync_read
 vfs_read
 sys_read
 syscall_call
 read
 + 1.43% wget [kernel.kallsyms] [k] ext3_get_blocks_handle
 + 1.42% wget [kernel.kallsyms] [k] tcp_recvmmsg
Press '?' for help on key bindings

```

At the lowest level of detail, we can dive down to the assembly level and see which instructions caused the most overhead in a function. Pressing 'enter' on the 'udhpcp\_main' function, we're again presented with a menu:

```

trz@empanada:~
File Edit View Search Terminal Help
Annotate udhpcp_main
Zoom into wget(1241) thread
Zoom out of busybox DS0
Browse map details
Exit
ESC: exit, ENTER| ->: Select option

```

Selecting 'Annotate udhpcp\_main', we get a detailed listing of percentages by instruction for the udhpcp\_main function. From the display, we can see that over 50% of the time spent in this function is taken up by a couple tests and the move of a constant (1) to a register:

```

trz@empanada:~
File Edit View Search Terminal Help
udhcpc main
3.70 : 805307a: test %cl,%cl
7.41 : 805307c: jne 80531aa <fchmod@plt+0x6fda>
0.00 : 8053082: lea 0x28(%esp),%esi
0.00 : 8053086: mov %esi,0x18(%esp)
0.00 : 805308a: mov $0x1000,%edi
0.00 : 805308f: mov 0x48(%ebx),%esi
0.00 : 8053092: mov 0x18(%esp),%eax
3.70 : 8053096: mov $0x3e8,%ecx
0.00 : 805309b: mov $0x1,%edx
0.00 : 80530a0: call 80ac1b5 <fchmod@plt+0x5ffe5>
11.11 : 80530a5: test %eax,%eax
0.00 : 80530a7: je 80531e2 <fchmod@plt+0x7012>
0.00 : 80530ad: mov %ebp,(%esp)
3.70 : 80530b0: call 804bf10 <clearerr@plt>
0.00 : 80530b5: mov 0x80d3988,%esi
0.00 : 80530bb: movl $0x0,(%esi)
0.00 : 80530c1: mov 0x1c(%esp),%esi
0.00 : 80530c5: mov %ebp,0xc(%esp)
3.70 : 80530c9: mov %edi,0x8(%esp)
0.00 : 80530cd: movl $0x1,0x4(%esp)
0.00 : 80530d5: mov %esi,(%esp)
0.00 : 80530d8: call 804b5c0 <fread@plt>
29.63 : 80530dd: test %eax,%eax
0.00 : 80530df: mov %eax,%esi
0.00 : 80530e1: jg 80530f8 <fchmod@plt+0x6f28>
0.00 : 80530e3: mov 0x80d3988,%esi
0.00 : 80530e9: cmpl $0xb,(%esi)
0.00 : 80530ec: jne 8053210 <fchmod@plt+0x7040>
0.00 : 80530f2: movzbl 0x55(%ebx),%ecx
0.00 : 80530f6: jmp 805307a <fchmod@plt+0x6eaa>
0.00 : 80530f8: mov 0x4c(%ebx),%eax
0.00 : 80530fb: mov %esi,%ecx
0.00 : 80530fd: mov %esi,%edi
0.00 : 80530ff: mov 0x1c(%esp),%edx
0.00 : 8053103: sar $0x1f,%edi
0.00 : 8053106: call 80862ea <fchmod@plt+0x3a11a>
14.81 : 805310b: mov $0x1,%eax
0.00 : 8053110: add %esi,0x10(%ebx)
3.70 : 8053113: adc %edi,0x14(%ebx)
0.00 : 8053116: call 808bfff <fchmod@plt+0x3fe2f>
0.00 : 805311b: movzbl 0x55(%ebx),%ecx
3.70 : 805311f: test %cl,%cl
0.00 : 8053121: je 805308a <fchmod@plt+0x6eba>
0.00 : 8053127: mov (%ebx),%eax
<-/ESC: Exit, TAB/shift+TAB: Cycle hot lines, H: Go to hottest line, ->/ENTER: L

```

As a segue into tracing, let's try another profile using a different counter, something other than the default 'cycles'.

The tracing and profiling infrastructure in Linux has become unified in a way that allows us to use the same tool with a completely different set of counters, not just the standard hardware counters that traditional tools have had to restrict themselves to (of course the traditional tools can also make use of the expanded possibilities now available to them, and in some cases have, as mentioned previously).

We can get a list of the available events that can be used to profile a workload via 'perf list':

```

root@crownbay:~# perf list

List of pre-defined events (to be used in -e):
cpu-cycles OR cycles [Hardware event]
stalled-cycles-frontend OR idle-cycles-frontend [Hardware event]
stalled-cycles-backend OR idle-cycles-backend [Hardware event]
instructions [Hardware event]
cache-references [Hardware event]
cache-misses [Hardware event]
branch-instructions OR branches [Hardware event]
branch-misses [Hardware event]
bus-cycles [Hardware event]
ref-cycles [Hardware event]

cpu-clock [Software event]
task-clock [Software event]
page-faults OR faults [Software event]
minor-faults [Software event]
major-faults [Software event]
context-switches OR cs [Software event]
cpu-migrations OR migrations [Software event]
alignment-faults [Software event]
emulation-faults [Software event]

L1-dcache-loads [Hardware cache event]
L1-dcache-load-misses [Hardware cache event]
L1-dcache-prefetch-misses [Hardware cache event]
L1-icache-loads [Hardware cache event]
L1-icache-load-misses [Hardware cache event]
.
.
rNNN [Raw hardware event descriptor]
cpu/t1=v1[,t2=v2,t3 ...]/modifier [Raw hardware event descriptor]
(see 'perf list --help' on how to encode it)

mem:<addr>[:access] [Hardware breakpoint]

sunrpc:rpc_call_status [Tracepoint event]
sunrpc:rpc_bind_status [Tracepoint event]
sunrpc:rpc_connect_status [Tracepoint event]
sunrpc:rpc_task_begin [Tracepoint event]

```

|                                          |                    |
|------------------------------------------|--------------------|
| skb:kfree_skb                            | [Tracepoint event] |
| skb:consume_skb                          | [Tracepoint event] |
| skb:skb_copy_datagram_iovec              | [Tracepoint event] |
| net:net_dev_xmit                         | [Tracepoint event] |
| net:net_dev_queue                        | [Tracepoint event] |
| net:netif_receive_skb                    | [Tracepoint event] |
| net:netif_rx                             | [Tracepoint event] |
| napi:napi_poll                           | [Tracepoint event] |
| sock:sock_rcvqueue_full                  | [Tracepoint event] |
| sock:sock_exceed_buf_limit               | [Tracepoint event] |
| udp:udp_fail_queue_rcv_skb               | [Tracepoint event] |
| hda:hda_send_cmd                         | [Tracepoint event] |
| hda:hda_get_response                     | [Tracepoint event] |
| hda:hda_bus_reset                        | [Tracepoint event] |
| scsi:scsi_dispatch_cmd_start             | [Tracepoint event] |
| scsi:scsi_dispatch_cmd_error             | [Tracepoint event] |
| scsi:scsi_ah_wakeupt                     | [Tracepoint event] |
| drm:drm_vblank_event                     | [Tracepoint event] |
| drm:drm_vblank_event_queued              | [Tracepoint event] |
| drm:drm_vblank_event_delivered           | [Tracepoint event] |
| random:mix_pool_bytes                    | [Tracepoint event] |
| random:mix_pool_bytes_nolock             | [Tracepoint event] |
| random:credit_entropy_bits               | [Tracepoint event] |
| gpio:gpio_direction                      | [Tracepoint event] |
| gpio:gpio_value                          | [Tracepoint event] |
| block:block_rq_abort                     | [Tracepoint event] |
| block:block_rq_requeue                   | [Tracepoint event] |
| block:block_rq_issue                     | [Tracepoint event] |
| block:block_bio_bounce                   | [Tracepoint event] |
| block:block_bio_complete                 | [Tracepoint event] |
| block:block_bio_backmerge                | [Tracepoint event] |
| .                                        |                    |
| writeback:writeback_wake_thread          | [Tracepoint event] |
| writeback:writeback_wake_forker_thread   | [Tracepoint event] |
| writeback:writeback_bdi_register         | [Tracepoint event] |
| .                                        |                    |
| writeback:writeback_single_inode_requeue | [Tracepoint event] |
| writeback:writeback_single_inode         | [Tracepoint event] |
| kmem:kmalloc                             | [Tracepoint event] |
| kmem:kmem_cache_alloc                    | [Tracepoint event] |
| kmem:mm_page_alloc                       | [Tracepoint event] |
| kmem:mm_page_alloc_zone_locked           | [Tracepoint event] |
| kmem:mm_page_cpu_drain                   | [Tracepoint event] |
| kmem:mm_page_alloc_extfrag               | [Tracepoint event] |
| vmscan:mm_vm_scan_kswapd_sleep           | [Tracepoint event] |
| vmscan:mm_vm_scan_kswapd_wake            | [Tracepoint event] |
| vmscan:mm_vm_scan_wakeup_kswapd          | [Tracepoint event] |
| vmscan:mm_vm_scan_direct_reclaim_begin   | [Tracepoint event] |
| .                                        |                    |
| module:module_get                        | [Tracepoint event] |
| module:module_put                        | [Tracepoint event] |
| module:module_request                    | [Tracepoint event] |
| sched:sched_kthread_stop                 | [Tracepoint event] |
| sched:sched_wakeupt                      | [Tracepoint event] |
| sched:sched_wakeupt_new                  | [Tracepoint event] |
| sched:sched_process_fork                 | [Tracepoint event] |
| sched:sched_process_exec                 | [Tracepoint event] |
| sched:sched_stat_runtime                 | [Tracepoint event] |
| rcu:rcu_utilization                      | [Tracepoint event] |
| workqueue:workqueue_queue_work           | [Tracepoint event] |
| workqueue:workqueue_execute_end          | [Tracepoint event] |
| signal:signal_generate                   | [Tracepoint event] |
| signal:signal_deliver                    | [Tracepoint event] |
| timer:timer_init                         | [Tracepoint event] |
| timer:timer_start                        | [Tracepoint event] |
| timer:hrtimer_cancel                     | [Tracepoint event] |
| timer:itimer_state                       | [Tracepoint event] |
| timer:itimer_expire                      | [Tracepoint event] |
| irq:irq_handler_entry                    | [Tracepoint event] |
| irq:irq_handler_exit                     | [Tracepoint event] |
| irq:softirq_entry                        | [Tracepoint event] |
| irq:softirq_exit                         | [Tracepoint event] |
| irq:softirq_raise                        | [Tracepoint event] |
| printk:console                           | [Tracepoint event] |
| task:task_newtask                        | [Tracepoint event] |
| task:task_rename                         | [Tracepoint event] |
| syscalls:sys_enter_socketcall            | [Tracepoint event] |
| syscalls:sys_exit_socketcall             | [Tracepoint event] |
| .                                        |                    |
| .                                        |                    |
| syscalls:sys_enter_unshare               | [Tracepoint event] |
| syscalls:sys_exit_unshare                | [Tracepoint event] |
| raw_syscalls:sys_enter                   | [Tracepoint event] |
| raw_syscalls:sys_exit                    | [Tracepoint event] |

**Tying it Together:** These are exactly the same set of events defined by the trace event subsystem and exposed by ftrace/tracecmd/kernelshark as files in /sys/kernel/debug/tracing/events, by SystemTap as kernel.trace("tracepoint\_name") and (partially) accessed by LTTng.

Only a subset of these would be of interest to us when looking at this workload, so let's choose the most likely subsystems (identified by the string before the colon in the Tracepoint events) and do a 'perf stat' run using only those wildcarded subsystems:

```
root@crownbay:~# perf stat -e skb:* -e net:* -e napi:* -e sched:* -e workqueue:* -e irq:* -e syscalls:* wget http://download
Performance counter stats for 'wget http://downloads.yoctoproject.org/mirror/sources/linux-2.6.19.2.tar.bz2':
```

23323 skb:kfree\_skb

```

0 skb:consume_skb
49897 skb:skb_copy_datagram_iovec
6217 net:net_dev_xmit
6217 net:net_dev_queue
7962 net:netif_receive_skb
2 net:netif_rx
8340 napi:napi_poll
0 sched:sched_kthread_stop
0 sched:sched_kthread_stop_ret
3749 sched:sched_wakeup
0 sched:sched_wakeup_new
0 sched:sched_switch
29 sched:sched_migrate_task
0 sched:sched_process_free
1 sched:sched_process_exit
0 sched:sched_wait_task
0 sched:sched_process_wait
0 sched:sched_process_fork
1 sched:sched_process_exec
0 sched:sched_stat_wait
2106519415641 sched:sched_stat_sleep
0 sched:sched_stat_iowait
147453613 sched:sched_stat_blocked
12903026955 sched:sched_stat_runtime
0 sched:sched_pi_setprio
3574 workqueue:workqueue_queue_work
3574 workqueue:workqueue_activate_work
0 workqueue:workqueue_execute_start
0 workqueue:workqueue_execute_end
16631 irq:irq_handler_entry
16631 irq:irq_handler_exit
28521 irq:softirq_entry
28521 irq:softirq_exit
28728 irq:softirq_raise
1 syscalls:sys_enter_sendmmsg
1 syscalls:sys_exit_sendmmsg
0 syscalls:sys_enter_recvmmsg
0 syscalls:sys_exit_recvmmsg
14 syscalls:sys_enter_socketcall
14 syscalls:sys_exit_socketcall
.
.
.
16965 syscalls:sys_enter_read
16965 syscalls:sys_exit_read
12854 syscalls:sys_enter_write
12854 syscalls:sys_exit_write
.
.
.

```

58.029710972 seconds time elapsed



Let's pick one of these tracepoints and tell perf to do a profile using it as the sampling event:

```
root@crownbay:~# perf record -g -e sched:sched_wakeup wget http://downloads.yoctoproject.org/mirror/sources/linux-2.6.19.1
```



```

trz@empanada:~
File Edit View Search Terminal Help
Events: 2K sched:sched_wakeup
- 100.00% wget [kernel.kallsyms] [k] ttwu_do_wakeup
- ttwu_do_wakeup
- 86.91% ttwu_do_activate.constprop.86
- try_to_wake_up
+ 91.34% wake_up_process
- 8.66% default_wake_function
+ 81.53% __wake_up_common
+ 17.20% autoremove_wake_function
- 1.27% pollwake
- __wake_up_common
- 50.00% __wake_up_sync_key
sock_def_readable
sock_queue_rcv_skb
udp_queue_rcv_skb
udp_queue_rcv_skb
udp4_lib_rcv
udp_rcv
ip_local_deliver_finish
ip_local_deliver
ip_rcv_finish
ip_rcv
__netif_receive_skb
process_backlog
net_rx_action
do_softirq
sendmmsg
+ 50.00% __wake_up
- 13.09% try_to_wake_up
- default_wake_function
+ 52.01% __wake_up_common
- 47.99% pollwake
__wake_up_common
__wake_up_sync_key
sock_def_readable
tcp_rcv_established
tcp_v4_do_rcv
tcp_v4_rcv
ip_local_deliver_finish
ip_local_deliver
ip_rcv_finish
ip_rcv
__netif_receive_skb
netif_receive_skb
napi_gro_complete
napi_complete
pch_gbe_napi_poll
net_rx_action
do_softirq
poll
Press '?' for help on key bindings

```

The screenshot above shows the results of running a profile using `sched:sched_switch` tracepoint, which shows the relative costs of various paths to `sched_wakeup` (note that `sched_wakeup` is the name of the tracepoint - it's actually defined just inside `ttwu_do_wakeup()`, which accounts for the function name actually displayed in the profile:

```

/*
 * Mark the task runnable and perform wakeup-preemption.
 */
static void
ttwu_do_wakeup(struct rq *rq, struct task_struct *p, int wake_flags)
{
 trace_sched_wakeup(p, true);
 .
 .
}

```

A couple of the more interesting callchains are expanded and displayed above, basically some network receive paths that presumably end up waking up `wget` (`busybox`) when network data is ready.

Note that because tracepoints are normally used for tracing, the default sampling period for tracepoints is 1 i.e. for tracepoints `perf` will sample on every event occurrence (this can be changed using the `-c` option). This is in contrast to hardware counters such as for example the default 'cycles' hardware counter used for normal profiling, where sampling periods are much higher (in the thousands) because profiling should have as low an overhead as possible and sampling on every cycle would be prohibitively expensive.

### 19.1.2.2. Using `perf` to do Basic Tracing

Profiling is a great tool for solving many problems or for getting a high-level view of what's going on with a workload or across the system. It is however by definition an approximation, as suggested by the most prominent word associated with it, 'sampling'. On the one hand, it allows a representative picture of what's going on in the system to be cheaply taken, but on the other hand, that cheapness limits its utility when that data suggests a need to 'dive down' more deeply to discover what's really going on. In such cases, the only way to see what's really going on is to be able to look at (or summarize more intelligently) the individual steps that go into the higher-level behavior exposed by the coarse-grained profiling data.

As a concrete example, we can trace all the events we think might be applicable to our workload:

```

root@crownbay:~# perf record -g -e skb:* -e net:* -e napi:* -e sched:sched_switch -e sched:sched_wakeup -e irq:*
-e syscalls:sys_enter_read -e syscalls:sys_exit_read -e syscalls:sys_enter_write -e syscalls:sys_exit_write
wget http://downloads.yoctoproject.org/mirror/sources/linux-2.6.19.2.tar.bz2

```

We can look at the raw trace output using '`perf script`' with no arguments:



```
root@crownbay:~# perf script
```

```
perf 1262 [000] 11624.857082: sys_exit_read: 0x0
perf 1262 [000] 11624.857193: sched_wakeup: comm=migration/0 pid=6 prio=0 success=1 target_cpu=000
wget 1262 [001] 11624.858021: softirq_raise: vec=1 [action=TIMER]
wget 1262 [001] 11624.858074: softirq_entry: vec=1 [action=TIMER]
wget 1262 [001] 11624.858081: softirq_exit: vec=1 [action=TIMER]
wget 1262 [001] 11624.858166: sys_enter_read: fd: 0x0003, buf: 0xbfb82c940, count: 0x0200
wget 1262 [001] 11624.858177: sys_exit_read: 0x200
wget 1262 [001] 11624.858878: kfree_skb: skbaddr=0xeb248d80 protocol=0 location=0xc15a5308
wget 1262 [001] 11624.858945: kfree_skb: skbaddr=0xeb248d80 protocol=0 location=0xc15a5308
wget 1262 [001] 11624.859020: softirq_raise: vec=1 [action=TIMER]
wget 1262 [001] 11624.859076: softirq_entry: vec=1 [action=TIMER]
wget 1262 [001] 11624.859083: softirq_exit: vec=1 [action=TIMER]
wget 1262 [001] 11624.859167: sys_enter_read: fd: 0x0003, buf: 0xb7720000, count: 0x0400
wget 1262 [001] 11624.859192: sys_exit_read: 0x1d7
wget 1262 [001] 11624.859228: sys_enter_read: fd: 0x0003, buf: 0xb7720000, count: 0x0400
wget 1262 [001] 11624.859233: sys_exit_read: 0x0
wget 1262 [001] 11624.859573: sys_enter_read: fd: 0x0003, buf: 0xbfb82c580, count: 0x0200
wget 1262 [001] 11624.859584: sys_exit_read: 0x200
wget 1262 [001] 11624.859864: sys_enter_read: fd: 0x0003, buf: 0xb7720000, count: 0x0400
wget 1262 [001] 11624.859888: sys_exit_read: 0x400
wget 1262 [001] 11624.859935: sys_enter_read: fd: 0x0003, buf: 0xb7720000, count: 0x0400
wget 1262 [001] 11624.859944: sys_exit_read: 0x400
```

This gives us a detailed timestamped sequence of events that occurred within the workload with respect to those events.

In many ways, profiling can be viewed as a subset of tracing - theoretically, if you have a set of trace events that's sufficient to capture all the important aspects of a workload, you can derive any of the results or views that a profiling run can.

Another aspect of traditional profiling is that while powerful in many ways, it's limited by the granularity of the underlying data. Profiling tools offer various ways of sorting and presenting the sample data, which make it much more useful and amenable to user experimentation, but in the end it can't be used in an open-ended way to extract data that just isn't present as a consequence of the fact that conceptually, most of it has been thrown away.

Full-blown detailed tracing data does however offer the opportunity to manipulate and present the information collected during a tracing run in an infinite variety of ways.

Another way to look at it is that there are only so many ways that the 'primitive' counters can be used on their own to generate interesting output; to get anything more complicated than simple counts requires some amount of additional logic, which is typically very specific to the problem at hand. For example, if we wanted to make use of a 'counter' that maps to the value of the time difference between when a process was scheduled to run on a processor and the time it actually ran, we wouldn't expect such a counter to exist on its own, but we could derive one called say 'wakeup\_latency' and use it to extract a useful view of that metric from trace data. Likewise, we really can't figure out from standard profiling tools how much data every process on the system reads and writes, along with how many of those reads and writes fail completely. If we have sufficient trace data, however, we could with the right tools easily extract and present that information, but we'd need something other than pre-canned profiling tools to do that.

Luckily, there is a general-purpose way to handle such needs, called 'programming languages'. Making programming languages easily available to apply to such problems given the specific format of data is called a 'programming language binding' for that data and language. Perf supports two programming language bindings, one for Python and one for Perl.

**Tying it Together:** Language bindings for manipulating and aggregating trace data are of course not a new idea. One of the first projects to do this was IBM's DProbes dpcc compiler, an ANSI C compiler which targeted a low-level assembly language running on an in-kernel interpreter on the target system. This is exactly analogous to what Sun's DTrace did, except that DTrace invented its own language for the purpose. Systemtap, heavily inspired by DTrace, also created its own one-off language, but rather than running the product on an in-kernel interpreter, created an elaborate compiler-based machinery to translate its language into kernel modules written in C.

Now that we have the trace data in perf.data, we can use 'perf script -g' to generate a skeleton script with handlers for the read/write entry/exit events we recorded:

```
root@crownbay:~# perf script -g python
generated Python script: perf-script.py
```

The skeleton script simply creates a python function for each event type in the perf.data file. The body of each function simply prints the event name along with its parameters. For example:

```
def net__netif_rx(event_name, context, common_cpu,
 common_secs, common_nsecs, common_pid, common_comm,
 skbaddr, len, name):
 print_header(event_name, common_cpu, common_secs, common_nsecs,
 common_pid, common_comm)

 print "skbaddr=%u, len=%u, name=%s\n" % (skbaddr, len, name),
```

We can run that script directly to print all of the events contained in the perf.data file:

```
root@crownbay:~# perf script -s perf-script.py

in trace_begin
syscalls__sys_exit_read 0 11624.857082795 1262 perf nr=3, ret=0
sched__sched_wakeup 0 11624.857193498 1262 perf comm=migration/0, pid=6, prio=0, success=1
irq__softirq_raise 1 11624.858021635 1262 wget vec=TIMER
irq__softirq_entry 1 11624.858074075 1262 wget vec=TIMER
irq__softirq_exit 1 11624.858081389 1262 wget vec=TIMER
syscalls__sys_enter_read 1 11624.858166434 1262 wget nr=3, fd=3, buf=3213019456, count=512
syscalls__sys_exit_read 1 11624.858177924 1262 wget nr=3, ret=512
skb__kfree_skb 1 11624.858878188 1262 wget skbaddr=3945041280, location=3243922184, pi
skb__kfree_skb 1 11624.858945608 1262 wget skbaddr=3945037824, location=3243922184, pi
irq__softirq_raise 1 11624.859020942 1262 wget vec=TIMER
irq__softirq_entry 1 11624.859076935 1262 wget vec=TIMER
irq__softirq_exit 1 11624.859083469 1262 wget vec=TIMER
syscalls__sys_enter_read 1 11624.859167565 1262 wget nr=3, fd=3, buf=3077701632, count=1024
syscalls__sys_exit_read 1 11624.859192533 1262 wget nr=3, ret=471
syscalls__sys_enter_read 1 11624.859228072 1262 wget nr=3, fd=3, buf=3077701632, count=1024
```

|                          |                   |           |                             |            |
|--------------------------|-------------------|-----------|-----------------------------|------------|
| syscalls__sys_exit_read  | 1 11624.859233707 | 1262 wget | nr=3, ret=0                 |            |
| syscalls__sys_enter_read | 1 11624.859573008 | 1262 wget | nr=3, fd=3, buf=3213018496, | count=512  |
| syscalls__sys_exit_read  | 1 11624.859584818 | 1262 wget | nr=3, ret=512               |            |
| syscalls__sys_enter_read | 1 11624.859864562 | 1262 wget | nr=3, fd=3, buf=3077701632, | count=1024 |
| syscalls__sys_exit_read  | 1 11624.859888770 | 1262 wget | nr=3, ret=1024              |            |
| syscalls__sys_enter_read | 1 11624.859935140 | 1262 wget | nr=3, fd=3, buf=3077701632, | count=1024 |
| syscalls__sys_exit_read  | 1 11624.859944032 | 1262 wget | nr=3, ret=1024              |            |

That in itself isn't very useful; after all, we can accomplish pretty much the same thing by simply running 'perf script' without arguments in the same directory as the perf.data file.

We can however replace the print statements in the generated function bodies with whatever we want, and thereby make it infinitely more useful.

As a simple example, let's just replace the print statements in the function bodies with a simple function that does nothing but increment a per-event count. When the program is run against a perf.data file, each time a particular event is encountered, a tally is incremented for that event. For example:

```
def net__netif_rx(event_name, context, common_cpu,
 common_secs, common_nsecs, common_pid, common_comm,
 skbaddr, len, name):
 inc_counts(event_name)
```

Each event handler function in the generated code is modified to do this. For convenience, we define a common function called inc\_counts() that each handler calls; inc\_counts() simply tallies a count for each event using the 'counts' hash, which is a specialized hash function that does Perl-like autovivification, a capability that's extremely useful for kinds of multi-level aggregation commonly used in processing traces (see perf's documentation on the Python language binding for details):

```
counts = autodict()

def inc_counts(event_name):
 try:
 counts[event_name] += 1
 except TypeError:
 counts[event_name] = 1
```

Finally, at the end of the trace processing run, we want to print the result of all the per-event tallies. For that, we use the special 'trace\_end()' function:

```
def trace_end():
 for event_name, count in counts.iteritems():
 print "%-40s %10s\n" % (event_name, count)
```

The end result is a summary of all the events recorded in the trace:

|                              |       |
|------------------------------|-------|
| skb__skb_copy_datagram_iovec | 13148 |
| irq__softirq_entry           | 4796  |
| irq__irq_handler_exit        | 3805  |
| irq__softirq_exit            | 4795  |
| syscalls__sys_enter_write    | 8990  |
| net__net_dev_xmit            | 652   |
| skb__kfree_skb               | 4047  |
| sched__sched_wakeup          | 1155  |
| irq__irq_handler_entry       | 3804  |
| irq__softirq_raise           | 4799  |
| net__net_dev_queue           | 652   |
| syscalls__sys_enter_read     | 17599 |
| net__netif_receive_skb       | 1743  |
| syscalls__sys_exit_read      | 17598 |
| net__netif_rx                | 2     |
| napi__napi_poll              | 1877  |
| syscalls__sys_exit_write     | 8990  |

Note that this is pretty much exactly the same information we get from 'perf stat', which goes a little way to support the idea mentioned previously that given the right kind of trace data, higher-level profiling-type summaries can be derived from it.

Documentation on using the '[perf script](#)' [python binding](#).

### 19.1.2.3. System-Wide Tracing and Profiling¶

The examples so far have focused on tracing a particular program or workload - in other words, every profiling run has specified the program to profile in the command-line e.g. 'perf record wget ...'.

It's also possible, and more interesting in many cases, to run a system-wide profile or trace while running the workload in a separate shell.

To do system-wide profiling or tracing, you typically use the -a flag to 'perf record'.

To demonstrate this, open up one window and start the profile using the -a flag (press Ctrl-C to stop tracing):

```
root@crownbay:~# perf record -g -a
^C[perf record: Woken up 6 times to write data]
[perf record: Captured and wrote 1.400 MB perf.data (~61172 samples)]
```

In another window, run the wget test:

```
root@crownbay:~# wget http://downloads.yoctoproject.org/mirror/sources/linux-2.6.19.2.tar.bz2
Connecting to downloads.yoctoproject.org (140.211.169.59:80)
linux-2.6.19.2.tar.b 100% |*****| 41727k 0:00:00 ETA
```

Here we see entries not only for our wget load, but for other processes running on the system as well:

```

trz@empanada:~/kdev/tip
File Edit View Search Terminal Help
Events: 11K cycles
+ 8.05% swapper [kernel.kallsyms] [k] intel_idle
- 6.04% Xorg libc-2.16.so [.] memset
- memset
- 58.17% SGXQueueTransfer
 0xb767fb
 - PVR2DBlt
 70.54% 0xb76abf
 29.46% 0xb76ac1
 - 41.83% 0xb767f6
 PVR2DBlt
 0xb76abf
+ 3.61% swapper [kernel.kallsyms] [k] read_hpet
+ 1.93% swapper [kernel.kallsyms] [k] ioread32
+ 1.21% Xorg [kernel.kallsyms] [k] ohci_irq
+ 0.73% swapper [kernel.kallsyms] [k] debug_smp_p
+ 0.70% swapper [kernel.kallsyms] [k] menu_select
+ 0.69% swapper [kernel.kallsyms] [k] ohci_irq
+ 0.57% swapper [kernel.kallsyms] [k] sub_preempt
- 0.53% wget [kernel.kallsyms] [k] memset
- memset
+ 88.85% _block_write_begin
+ 8.22% alloc_buffer_head
- 1.65% kmem_cache_alloc
 alloc_buffer_head
 alloc_page_buffers
 create_empty_buffers
 _block_write_begin
 ext4_da_write_begin
 generic_file_buffered_write
 __generic_file_aio_write
 generic_file_aio_write
 ext4_file_write
 do_sync_write
 vfs_write
 sys_write
 syscall_call
 __GI___libc_write
+ 1.27% kmem_cache_alloc_trace
+ 0.51% swapper [kernel.kallsyms] [k] add_preempt
+ 0.49% Xorg [kernel.kallsyms] [k] read_hpet
+ 0.49% Xorg [emgd] [k] igd_alter_c
+ 0.48% wget [kernel.kallsyms] [k] read_hpet
+ 0.47% wget [kernel.kallsyms] [k] __copy_to_u
+ 0.45% swapper [kernel.kallsyms] [k] _raw_spin_l
+ 0.45% wget [kernel.kallsyms] [k] sub_preempt
Press '?' for help on key bindings

```

In the snapshot above, we can see callchains that originate in libc, and a callchain from Xorg that demonstrates that we're using a proprietary X driver in userspace (notice the presence of 'PVR' and some other unresolvable symbols in the expanded Xorg callchain).

Note also that we have both kernel and userspace entries in the above snapshot. We can also tell perf to focus on userspace but providing a modifier, in this case 'u', to the 'cycles' hardware counter when we record a profile:

```

root@crownbay:~# perf record -g -a -e cycles:u
^C[perf record: Woken up 2 times to write data]
[perf record: Captured and wrote 0.376 MB perf.data (~16443 samples)]

```

```

trz@empanada:~/kdev/tip
File Edit View Search Terminal Help
Events: 4K cycles
- 12.84% wget [kernel.kallsyms] [k] system_call
 - system_call
 39.73% __read_nocancel
 20.38% __GI__libc_poll
 19.90% syscalls
 19.66% __GI__libc_write
 + 7.01% wget busybox [.] retrieve_file_data
 + 5.71% wget libc-2.16.so [.] _IO_file_xsgetn
 + 4.49% wget libc-2.16.so [.] _IO_fread
 + 4.36% wget libc-2.16.so [.] clearerr
 + 3.84% wget libc-2.16.so [.] __GI__libc_poll
 + 3.69% wget libc-2.16.so [.] __read_nocancel
 + 3.42% wget busybox [.] progress_meter
 + 3.16% wget libc-2.16.so [.] syscalls
 + 3.11% wget busybox [.] safe_poll
 + 3.04% wget libc-2.16.so [.] __underflow
 + 2.95% wget libc-2.16.so [.] _IO_file_underflow@@
 + 2.95% wget libc-2.16.so [.] __GI__libc_write
 + 2.55% wget busybox [.] bb_progress_update
 + 2.54% wget busybox [.] 0x00004f10
 - 2.32% wget libc-2.16.so [.] __x86.get_pc_thunk.bx
 - __x86.get_pc_thunk.bx
 - 68.83% fprintf
 progress_meter
 - 31.17% buffered_vfprintf
 fprintf
 progress_meter
 + 1.79% wget busybox [.] xwrite
 + 1.71% wget busybox [.] full write
 + 1.65% wget libc-2.16.so [.] _IO_file_read
 + 1.57% wget libc-2.16.so [.] _IO_sgetn
 + 1.43% wget busybox [.] safe_write
 + 1.09% wget libc-2.16.so [.] __GI__libc_read
 + 0.85% wget libc-2.16.so [.] _IO_switch_to_get_mo
 + 0.80% dropbear dropbearmulti [.] rijndael_ecb_encrypt
 + 0.78% perf perf [.] hex2u64
 + 0.71% dropbear dropbearmulti [.] md5_compress
 + 0.60% wget busybox [.] monotonic_sec
 + 0.57% dropbear libc-2.16.so [.] _int_free
 + 0.57% wget busybox [.] get_mono
 + 0.55% dropbear dropbearmulti [.] sha1_compress
 + 0.39% perf libc-2.16.so [.] __x86.get_pc_thunk.bx
 + 0.38% dropbear libc-2.16.so [.] _int_malloc
 + 0.36% wget libc-2.16.so [.] vfprintf
 + 0.36% wget libc-2.16.so [.] 0x000c9ab4
Press '?' for help on key bindings

```

Notice in the screenshot above, we see only userspace entries ([.])

Finally, we can press 'enter' on a leaf node and select the 'Zoom into DSO' menu item to show only entries associated with a specific DSO. In the screenshot below, we've zoomed into the 'libc' DSO which shows all the entries associated with the libc-xxx.so DSO.

```

trz@empanada:~/kdev/tip
File Edit View Search Terminal Help
Events: 1K cycles, DS0: libc-2.16.so
- 60.86% Xorg [.] memset
 - memset
 - 58.17% SGXQueueTransfer
 0xb767fb
 - PVR2DBlt
 70.54% 0xb76abf
 29.46% 0xb76ac1
 - 41.83% 0xb767f6
 PVR2DBlt
 0xb76abf
- 2.61% pcmanfm [.] strcmp
 - strcmp
 + 50.21% __gconv_find_transform
 49.79% __dcgettext
- 2.08% pcmanfm [.] _int_malloc
 - _int_malloc
- 1.92% Xorg [.] _int_malloc
 - _int_malloc
- 1.49% Xorg [.] memcpy
 - memcpy
 - 100.00% SGXQueueTransfer
 0xb767fb
 PVR2DBlt
 0xb76abf
+ 1.29% wget [.] __GI__libc_poll
+ 1.06% wget [.] __read_nocancel
+ 0.97% wget [.] _IO_file_xsgetn
+ 0.93% wget [.] clearerr
+ 0.90% wget [.] syscall
+ 0.79% pcmanfm [.] malloc
+ 0.73% leafpad [.] strcmp
+ 0.70% wget [.] __underflow
+ 0.68% Xorg [.] _int_free
+ 0.64% pcmanfm [.] memcpy
+ 0.61% wget [.] _IO_fread
+ 0.60% leafpad [.] _int_malloc
+ 0.59% Xorg [.] free
+ 0.49% Xorg [.] __select
+ 0.49% pcmanfm [.] __x86.get_pc_thunk.bx
+ 0.47% pcmanfm [.] memset
+ 0.46% slant [.] strcmp
+ 0.44% wget [.] __x86.get_pc_thunk.bx
+ 0.44% matchbox-deskto [.] memcpy
+ 0.39% slant [.] __x86.get_pc_thunk.bx
+ 0.38% matchbox-deskto [.] _int_malloc
Press '?' for help on key bindings

```

We can also use the system-wide -a switch to do system-wide tracing. Here we'll trace a couple of scheduler events:

```

root@crownbay:~# perf record -a -e sched:sched_switch -e sched:sched_wakeup
^C[perf record: Woken up 38 times to write data]
[perf record: Captured and wrote 9.780 MB perf.data (~427299 samples)]

```

We can look at the raw output using 'perf script' with no arguments:

```

root@crownbay:~# perf script

perf 1383 [001] 6171.460045: sched_wakeup: comm=kworker/1:1 pid=21 prio=120 success=1 target_cpu=001
perf 1383 [001] 6171.460066: sched_switch: prev_comm=perf prev_pid=1383 prev_prio=120 prev_state=R ==> next
kworker/1:1 21 [001] 6171.460093: sched_switch: prev_comm=kworker/1:1 prev_pid=21 prev_prio=120 prev_state=S ==> i
swapper 0 [000] 6171.468063: sched_wakeup: comm=kworker/0:3 pid=1209 prio=120 success=1 target_cpu=000
swapper 0 [000] 6171.468107: sched_switch: prev_comm=swapper/0 prev_pid=0 prev_prio=120 prev_state=R ==> nex
kworker/0:3 1209 [000] 6171.468143: sched_switch: prev_comm=kworker/0:3 prev_pid=1209 prev_prio=120 prev_state=S ==
perf 1383 [001] 6171.470039: sched_wakeup: comm=kworker/1:1 pid=21 prio=120 success=1 target_cpu=001
perf 1383 [001] 6171.470058: sched_switch: prev_comm=perf prev_pid=1383 prev_prio=120 prev_state=R ==> next
kworker/1:1 21 [001] 6171.470082: sched_switch: prev_comm=kworker/1:1 prev_pid=21 prev_prio=120 prev_state=S ==> i
perf 1383 [001] 6171.480035: sched_wakeup: comm=kworker/1:1 pid=21 prio=120 success=1 target_cpu=001

```

#### 19.1.2.3.1. Filtering

Notice that there are a lot of events that don't really have anything to do with what we're interested in, namely events that schedule 'perf' itself in and out or that wake perf up. We can get rid of those by using the '--filter' option - for each event we specify using -e, we can add a --filter after that to filter out trace events that contain fields with specific values:

```

root@crownbay:~# perf record -a -e sched:sched_switch --filter 'next_comm != perf && prev_comm != perf' -e sched:sched_wal
^C[perf record: Woken up 38 times to write data]
[perf record: Captured and wrote 9.688 MB perf.data (~423279 samples)]

```

```

root@crownbay:~# perf script

swapper 0 [000] 7932.162180: sched_switch: prev_comm=swapper/0 prev_pid=0 prev_prio=120 prev_state=R ==> nex
kworker/0:3 1209 [000] 7932.162236: sched_switch: prev_comm=kworker/0:3 prev_pid=1209 prev_prio=120 prev_state=S ==
perf 1407 [001] 7932.170048: sched_wakeup: comm=kworker/1:1 pid=21 prio=120 success=1 target_cpu=001
perf 1407 [001] 7932.180044: sched_wakeup: comm=kworker/1:1 pid=21 prio=120 success=1 target_cpu=001
perf 1407 [001] 7932.190038: sched_wakeup: comm=kworker/1:1 pid=21 prio=120 success=1 target_cpu=001
perf 1407 [001] 7932.200044: sched_wakeup: comm=kworker/1:1 pid=21 prio=120 success=1 target_cpu=001
perf 1407 [001] 7932.210044: sched_wakeup: comm=kworker/1:1 pid=21 prio=120 success=1 target_cpu=001
perf 1407 [001] 7932.220044: sched_wakeup: comm=kworker/1:1 pid=21 prio=120 success=1 target_cpu=001
swapper 0 [001] 7932.230111: sched_wakeup: comm=kworker/1:1 pid=21 prio=120 success=1 target_cpu=001
swapper 0 [001] 7932.230146: sched_switch: prev_comm=swapper/1 prev_pid=0 prev_prio=120 prev_state=R ==> nex
kworker/1:1 21 [001] 7932.230205: sched_switch: prev_comm=kworker/1:1 prev_pid=21 prev_prio=120 prev_state=S ==> i
swapper 0 [000] 7932.326109: sched_wakeup: comm=kworker/0:3 pid=1209 prio=120 success=1 target_cpu=000
swapper 0 [000] 7932.326171: sched_switch: prev_comm=swapper/0 prev_pid=0 prev_prio=120 prev_state=R ==> nex

```

```
kworker/0:3 1209 [000] 7932.326214: sched_switch: prev_comm=kworker/0:3 prev_pid=1209 prev_prio=120 prev_state=S ==
```

In this case, we've filtered out all events that have 'perf' in their 'comm' or 'comm\_prev' or 'comm\_next' fields. Notice that there are still events recorded for perf, but notice that those events don't have values of 'perf' for the filtered fields. To completely filter out anything from perf will require a bit more work, but for the purpose of demonstrating how to use filters, it's close enough.

**Tying it Together:** These are exactly the same set of event filters defined by the trace event subsystem. See the `ftace/tracecmd/kernelshark` section for more discussion about these event filters.

**Tying it Together:** These event filters are implemented by a special-purpose pseudo-interpreter in the kernel and are an integral and indispensable part of the perf design as it relates to tracing. kernel-based event filters provide a mechanism to precisely throttle the event stream that appears in user space, where it makes sense to provide bindings to real programming languages for postprocessing the event stream. This architecture allows for the intelligent and flexible partitioning of processing between the kernel and user space. Contrast this with other tools such as SystemTap, which does all of its processing in the kernel and as such requires a special project-defined language in order to accommodate that design, or LTTng, where everything is sent to userspace and as such requires a super-efficient kernel-to-userspace transport mechanism in order to function properly. While perf certainly can benefit from for instance advances in the design of the transport, it doesn't fundamentally depend on them. Basically, if you find that your perf tracing application is causing buffer I/O overruns, it probably means that you aren't taking enough advantage of the kernel filtering engine.

#### 19.1.2.4. Using Dynamic Tracepoints

perf isn't restricted to the fixed set of static tracepoints listed by 'perf list'. Users can also add their own 'dynamic' tracepoints anywhere in the kernel. For instance, suppose we want to define our own tracepoint on `do_fork()`. We can do that using the 'perf probe' perf subcommand:

```
root@crownbay:~# perf probe do_fork
Added new event:
 probe:do_fork (on do_fork)

You can now use it in all perf tools, such as:

perf record -e probe:do_fork -aR sleep 1
```

Adding a new tracepoint via 'perf probe' results in an event with all the expected files and format in `/sys/kernel/debug/tracing/events`, just the same as for static tracepoints (as discussed in more detail in the trace events subsystem section):

```
root@crownbay:/sys/kernel/debug/tracing/events/probe/do_fork# ls -al
drwxr-xr-x 2 root root 0 Oct 28 11:42 .
drwxr-xr-x 3 root root 0 Oct 28 11:42 ..
-rw-r--r-- 1 root root 0 Oct 28 11:42 enable
-rw-r--r-- 1 root root 0 Oct 28 11:42 filter
-r--r--r-- 1 root root 0 Oct 28 11:42 format
-r--r--r-- 1 root root 0 Oct 28 11:42 id

root@crownbay:/sys/kernel/debug/tracing/events/probe/do_fork# cat format
name: do_fork
ID: 944
format:
 field:unsigned short common_type; offset:0; size:2; signed:0;
 field:unsigned char common_flags; offset:2; size:1; signed:0;
 field:unsigned char common_preempt_count; offset:3; size:1; signed:0;
 field:int common_pid; offset:4; size:4; signed:1;
 field:int common_padding; offset:8; size:4; signed:1;

 field:unsigned long __probe_ip; offset:12; size:4; signed:0;

print fmt: "({%lx})", REC->__probe_ip
```

We can list all dynamic tracepoints currently in existence:

```
root@crownbay:~# perf probe -l
probe:do_fork (on do_fork)
probe:schedule (on schedule)
```

Let's record system-wide ('sleep 30' is a trick for recording system-wide but basically do nothing and then wake up after 30 seconds):

```
root@crownbay:~# perf record -g -a -e probe:do_fork sleep 30
[perf record: Woken up 1 times to write data]
[perf record: Captured and wrote 0.087 MB perf.data (~3812 samples)]
```

Using 'perf script' we can see each `do_fork` event that fired:

```
root@crownbay:~# perf script

=====
captured on: Sun Oct 28 11:55:18 2012
hostname : crownbay
os release : 3.4.11-yocto-standard
perf version : 3.4.11
arch : i686
nr_cpus online : 2
nr_cpus avail : 2
cpudesc : Intel(R) Atom(TM) CPU E660 @ 1.30GHz
cpuid : GenuineIntel,6,38,1
total memory : 1017184 kB
cmdline : /usr/bin/perf record -g -a -e probe:do_fork sleep 30
event : name = probe:do_fork, type = 2, config = 0x3b0, config1 = 0x0, config2 = 0x0, excl_usr = 0, excl_kern
 = 0, id = { 5, 6 }
HEADER_CPU_TOPOLOGY info available, use -I to display
=====
#
matchbox-deskto 1197 [001] 34211.378318: do fork: (c1028460)
```

```

matchbox-deskto 1295 [001] 34211.380388: do_fork: (c1028460)
pcmanfm 1296 [000] 34211.632350: do_fork: (c1028460)
pcmanfm 1296 [000] 34211.639917: do_fork: (c1028460)
matchbox-deskto 1197 [001] 34217.541603: do_fork: (c1028460)
matchbox-deskto 1299 [001] 34217.543584: do_fork: (c1028460)
gthumb 1300 [001] 34217.697451: do_fork: (c1028460)
gthumb 1300 [001] 34219.085734: do_fork: (c1028460)
gthumb 1300 [000] 34219.121351: do_fork: (c1028460)
gthumb 1300 [001] 34219.264551: do_fork: (c1028460)
pcmanfm 1296 [000] 34219.590380: do_fork: (c1028460)
matchbox-deskto 1197 [001] 34224.955965: do_fork: (c1028460)
matchbox-deskto 1306 [001] 34224.957972: do_fork: (c1028460)
matchbox-termin 1307 [000] 34225.038214: do_fork: (c1028460)
matchbox-termin 1307 [001] 34225.044218: do_fork: (c1028460)
matchbox-termin 1307 [000] 34225.046442: do_fork: (c1028460)
matchbox-deskto 1197 [001] 34237.112138: do_fork: (c1028460)
matchbox-deskto 1311 [001] 34237.114106: do_fork: (c1028460)
gaku 1312 [000] 34237.202388: do_fork: (c1028460)

```

And using 'perf report' on the same file, we can see the callgraphs from starting a few programs during those 30 seconds:

```

trz@empanada: ~/danny-meta-intel-release
File Edit View Search Terminal Help
Events: 19 probe:do_fork
- 42.11% matchbox-deskto [kernel.kallsyms] [k] do_fork
 do_fork
 ptregs_clone
 __libc_fork
 fork_exec_with_pipes
- 21.05% gthumb [kernel.kallsyms] [k] do_fork
 do_fork
 ptregs_clone
 __clone
 0
- 15.79% pcmanfm [kernel.kallsyms] [k] do_fork
 do_fork
 ptregs_clone
 __clone
 0
- 15.79% matchbox-termin [kernel.kallsyms] [k] do_fork
 do_fork
 ptregs_clone
 - 66.67% __libc_fork
 50.00% vte_ptty_initable_init
 50.00% fork_exec_with_pipes
 - 33.33% __clone
 0
- 5.26% gaku [kernel.kallsyms] [k] do_fork
 do_fork
 ptregs_clone
 __libc_fork
 fork
Press '?' for help on key bindings

```

**Tying it Together:** The trace events subsystem accommodate static and dynamic tracepoints in exactly the same way - there's no difference as far as the infrastructure is concerned. See the ftrace section for more details on the trace event subsystem.

**Tying it Together:** Dynamic tracepoints are implemented under the covers by kprobes and uprobes. kprobes and uprobes are also used by and in fact are the main focus of SystemTap.

### 19.1.3. Documentation¶

Online versions of the man pages for the commands discussed in this section can be found here:

- The '[perf stat](#)' manpage.
- The '[perf record](#)' manpage.
- The '[perf report](#)' manpage.
- The '[perf probe](#)' manpage.
- The '[perf script](#)' manpage.
- Documentation on using the '[perf script](#)' python binding.
- The top-level [perf\(1\)](#) manpage.

Normally, you should be able to invoke the man pages via perf itself e.g. 'perf help' or 'perf help record'.

However, by default Yocto doesn't install man pages, but perf invokes the man pages for most help functionality. This is a bug and is being addressed by a Yocto bug: [Bug\\_3388 - perf: enable man pages for basic 'help' functionality](#).

The man pages in text form, along with some other files, such as a set of examples, can be found in the 'perf' directory of the kernel tree:

```
tools/perf/Documentation
```

There's also a nice perf tutorial on the perf wiki that goes into more detail than we do here in certain areas: [Perf Tutorial](#)

## 19.2. ftrace¶

'ftrace' literally refers to the 'ftrace function tracer' but in reality this encompasses a number of related tracers along with the infrastructure that they all make use of.

### 19.2.1. Setup¶

For this section, we'll assume you've already performed the basic setup outlined in the General Setup section.

ftrace, trace-cmd, and kernelshark run on the target system, and are ready to go out-of-the-box - no additional setup is necessary. For the rest of this section we assume you've ssh'ed to the host and will be running ftrace on the target. kernelshark is a GUI application and if you use the '-X' option to ssh you can have the kernelshark GUI run on the target but display remotely on the host if you want.

### 19.2.2. Basic ftrace usage¶

'ftrace' essentially refers to everything included in the /tracing directory of the mounted debugfs filesystem (Yocto follows the standard convention and mounts it at /sys/kernel/debug). Here's a listing of all the files found in /sys/kernel/debug/tracing on a Yocto system:

```
root@sugarbay:/sys/kernel/debug/tracing# ls
README kprobe_events trace
available_events kprobe_profile trace_clock
available_filter_functions options trace_marker
available_tracers per_cpu trace_options
buffer_size_kb printk_formats trace_pipe
buffer_total_size_kb saved_cmdlines tracing_cpumask
current_tracer set_event tracing_enabled
dyn_ftrace_total_info set_ftrace_filter tracing_on
enabled_functions set_ftrace_notrace tracing_thresh
events set_ftrace_pid
free_buffer set_graph_function
```

The files listed above are used for various purposes - some relate directly to the tracers themselves, others are used to set tracing options, and yet others actually contain the tracing output when a tracer is in effect. Some of the functions can be guessed from their names, others need explanation; in any case, we'll cover some of the files we see here below but for an explanation of the others, please see the ftrace documentation.

We'll start by looking at some of the available built-in tracers.

cat'ing the 'available\_tracers' file lists the set of available tracers:

```
root@sugarbay:/sys/kernel/debug/tracing# cat available_tracers
blk function_graph function nop
```

The 'current\_tracer' file contains the tracer currently in effect:

```
root@sugarbay:/sys/kernel/debug/tracing# cat current_tracer
nop
```

The above listing of current\_tracer shows that the 'nop' tracer is in effect, which is just another way of saying that there's actually no tracer currently in effect.

echo'ing one of the available\_tracers into current\_tracer makes the specified tracer the current tracer:

```
root@sugarbay:/sys/kernel/debug/tracing# echo function > current_tracer
root@sugarbay:/sys/kernel/debug/tracing# cat current_tracer
function
```

The above sets the current tracer to be the 'function tracer'. This tracer traces every function call in the kernel and makes it available as the contents of the 'trace' file. Reading the 'trace' file lists the currently buffered function calls that have been traced by the function tracer:

```
root@sugarbay:/sys/kernel/debug/tracing# cat trace | less

tracer: function
#
entries-in-buffer/entries-written: 310629/766471 #P:8
#
_-----> irqs-off
/ _-----> need-resched
| / _-----> hardirq/softirq
|| / _-----> preempt-depth
||| / _-----> delay
#
TASK-PID CPU# TIMESTAMP FUNCTION
| | | | |
<idle>-0 [004] d..1 470.867169: ktime_get_real <-intel_idle
<idle>-0 [004] d..1 470.867170: getnstimeofday <-ktime_get_real
<idle>-0 [004] d..1 470.867171: ns_to_timeval <-intel_idle
<idle>-0 [004] d..1 470.867171: ns_to_timespec <-ns_to_timeval
<idle>-0 [004] d..1 470.867172: smp_apic_timer_interrupt <-apic_timer_interrupt
<idle>-0 [004] d..1 470.867172: native_apic_mem_write <-smp_apic_timer_interrupt
```



```

<idle>-0 [004] d..1 470.867172: irq_enter <-smp_apic_timer_interrupt
<idle>-0 [004] d..1 470.867172: rcu_irq_enter <-irq_enter
<idle>-0 [004] d..1 470.867173: rcu_idle_exit_common.isra.33 <-rcu_irq_enter
<idle>-0 [004] d..1 470.867173: local_bh_disable <-irq_enter
<idle>-0 [004] d..1 470.867173: add_preempt_count <-local_bh_disable
<idle>-0 [004] d.s1 470.867174: tick_check_idle <-irq_enter
<idle>-0 [004] d.s1 470.867174: tick_check_oneshot_broadcast <-tick_check_idle
<idle>-0 [004] d.s1 470.867174: ktime_get <-tick_check_idle
<idle>-0 [004] d.s1 470.867174: tick_nohz_stop_idle <-tick_check_idle
<idle>-0 [004] d.s1 470.867175: update_ts_time_stats <-tick_nohz_stop_idle
<idle>-0 [004] d.s1 470.867175: nr_iowait_cpu <-update_ts_time_stats
<idle>-0 [004] d.s1 470.867175: tick_do_update_jiffies64 <-tick_check_idle
<idle>-0 [004] d.s1 470.867175: _raw_spin_lock <-tick_do_update_jiffies64
<idle>-0 [004] d.s1 470.867176: add_preempt_count <-_raw_spin_lock
<idle>-0 [004] d.s2 470.867176: do_timer <-tick_do_update_jiffies64
<idle>-0 [004] d.s2 470.867176: _raw_spin_lock <-do_timer
<idle>-0 [004] d.s2 470.867176: add_preempt_count <-_raw_spin_lock
<idle>-0 [004] d.s3 470.867177: ntp_tick_length <-do_timer
<idle>-0 [004] d.s3 470.867177: _raw_spin_lock_irqsave <-ntp_tick_length
.
.
.

```

Each line in the trace above shows what was happening in the kernel on a given cpu, to the level of detail of function calls. Each entry shows the function called, followed by its caller (after the arrow).

The function tracer gives you an extremely detailed idea of what the kernel was doing at the point in time the trace was taken, and is a great way to learn about how the kernel code works in a dynamic sense.

**Tying it Together:** The ftrace function tracer is also available from within perf, as the ftrace: function tracepoint.

It is a little more difficult to follow the call chains than it needs to be - luckily there's a variant of the function tracer that displays the callchains explicitly, called the 'function\_graph' tracer:

```

root@sugarbay:/sys/kernel/debug/tracing# echo function_graph > current_tracer
root@sugarbay:/sys/kernel/debug/tracing# cat trace | less

```

```

tracer: function_graph

CPU DURATION FUNCTION CALLS
| | |
7) 0.046 us | pick_next_task_fair();
7) 0.043 us | pick_next_task_stop();
7) 0.042 us | pick_next_task_rt();
7) 0.032 us | pick_next_task_fair();
7) 0.030 us | pick_next_task_idle();
7) | _raw_spin_unlock_irq() {
7) 0.033 us | sub_preempt_count();
7) 0.258 us | }
7) 0.032 us | sub_preempt_count();
7) + 13.341 us | } /* __schedule */
7) 0.095 us | } /* sub_preempt_count */
7) | schedule() {
7) | __schedule() {
7) 0.060 us | add_preempt_count();
7) 0.044 us | rcu_note_context_switch();
7) | _raw_spin_lock_irq() {
7) 0.033 us | add_preempt_count();
7) 0.247 us | }
7) | idle_balance() {
7) | _raw_spin_unlock() {
7) 0.031 us | sub_preempt_count();
7) 0.246 us | }
7) | update_shares() {
7) 0.030 us | __rcu_read_lock();
7) 0.029 us | __rcu_read_unlock();
7) 0.484 us | }
7) 0.030 us | __rcu_read_lock();
7) | load_balance() {
7) | find_busiest_group() {
7) 0.031 us | idle_cpu();
7) 0.029 us | idle_cpu();
7) 0.035 us | idle_cpu();
7) 0.906 us | }
7) 1.141 us | }
7) 0.022 us | msec_to_jiffies();
7) | load_balance() {
7) | find_busiest_group() {
7) 0.031 us | idle_cpu();
.
.
.
4) 0.062 us | msec_to_jiffies();
4) 0.062 us | __rcu_read_unlock();
4) | _raw_spin_lock() {
4) 0.073 us | add_preempt_count();
4) 0.562 us | }
4) + 17.452 us | }
4) 0.108 us | put_prev_task_fair();
4) 0.102 us | pick_next_task_fair();
4) 0.084 us | pick_next_task_stop();
4) 0.075 us | pick_next_task_rt();
4) 0.062 us | pick_next_task_fair();
4) 0.066 us | pick_next_task_idle();

4) kworker-74 => <idle>-0

4) | finish_task_switch() {
4) | _raw_spin_unlock_irq() {

```

```

4) 0.100 us | sub_preempt_count();
4) 0.582 us | }
4) 1.105 us | }
4) 0.088 us | sub_preempt_count();
4) ! 100.066 us | }
.
.
.
3) sys_ioctl() {
3) 0.083 us | fget_light();
3) security_file_ioctl() {
3) 0.066 us | cap_file_ioctl();
3) 0.562 us | }
3) do_vfs_ioctl() {
3) drm_ioctl() {
3) 0.075 us | drm_ut_debug_printk();
3) i915_gem_pwrite_ioctl() {
3) i915_mutex_lock_interruptible() {
3) 0.070 us | mutex_lock_interruptible();
3) 0.570 us | }
3) drm_gem_object_lookup() {
3) _raw_spin_lock() {
3) 0.080 us | add_preempt_count();
3) 0.620 us | }
3) _raw_spin_unlock() {
3) 0.085 us | sub_preempt_count();
3) 0.562 us | }
3) 2.149 us | }
3) 0.133 us | i915_gem_object_pin();
3) i915_gem_object_set_to_gtt_domain() {
3) 0.065 us | i915_gem_object_flush_gpu_write_domain();
3) 0.065 us | i915_gem_object_wait_rendering();
3) 0.062 us | i915_gem_object_flush_cpu_write_domain();
3) 1.612 us | }
3) i915_gem_object_put_fence() {
3) 0.097 us | i915_gem_object_flush_fence.constprop.36();
3) 0.645 us | }
3) 0.070 us | add_preempt_count();
3) 0.070 us | sub_preempt_count();
3) 0.073 us | i915_gem_object_unpin();
3) 0.068 us | mutex_unlock();
3) 9.924 us | }
3) + 11.236 us | }
3) + 11.770 us | }
3) + 13.784 us | }
3) sys_ioctl() {

```

As you can see, the function\_graph display is much easier to follow. Also note that in addition to the function calls and associated braces, other events such as scheduler events are displayed in context. In fact, you can freely include any tracepoint available in the trace events subsystem described in the next section by simply enabling those events, and they'll appear in context in the function graph display. Quite a powerful tool for understanding kernel dynamics.

Also notice that there are various annotations on the left hand side of the display. For example if the total time it took for a given function to execute is above a certain threshold, an exclamation point or plus sign appears on the left hand side. Please see the trace documentation for details on all these fields.

### 19.2.3. The 'trace events' Subsystem¶

One especially important directory contained within the /sys/kernel/debug/tracing directory is the 'events' subdirectory, which contains representations of every tracepoint in the system. Listing out the contents of the 'events' subdirectory, we see mainly another set of subdirectories:

```

root@sugarbay:/sys/kernel/debug/tracing# cd events
root@sugarbay:/sys/kernel/debug/tracing/events# ls -al
drwxr-xr-x 38 root root 0 Nov 14 23:19 .
drwxr-xr-x 5 root root 0 Nov 14 23:19 ..
drwxr-xr-x 19 root root 0 Nov 14 23:19 block
drwxr-xr-x 32 root root 0 Nov 14 23:19 btrfs
drwxr-xr-x 5 root root 0 Nov 14 23:19 drm
-rw-r--r-- 1 root root 0 Nov 14 23:19 enable
drwxr-xr-x 40 root root 0 Nov 14 23:19 ext3
drwxr-xr-x 79 root root 0 Nov 14 23:19 ext4
drwxr-xr-x 14 root root 0 Nov 14 23:19 ftrace
drwxr-xr-x 8 root root 0 Nov 14 23:19 hda
-r--r--r-- 1 root root 0 Nov 14 23:19 header_event
-r--r--r-- 1 root root 0 Nov 14 23:19 header_page
drwxr-xr-x 25 root root 0 Nov 14 23:19 i915
drwxr-xr-x 7 root root 0 Nov 14 23:19 irq
drwxr-xr-x 12 root root 0 Nov 14 23:19 jbd
drwxr-xr-x 14 root root 0 Nov 14 23:19 jbd2
drwxr-xr-x 14 root root 0 Nov 14 23:19 kmem
drwxr-xr-x 7 root root 0 Nov 14 23:19 module
drwxr-xr-x 3 root root 0 Nov 14 23:19 napi
drwxr-xr-x 6 root root 0 Nov 14 23:19 net
drwxr-xr-x 3 root root 0 Nov 14 23:19 oom
drwxr-xr-x 12 root root 0 Nov 14 23:19 power
drwxr-xr-x 3 root root 0 Nov 14 23:19 printk
drwxr-xr-x 8 root root 0 Nov 14 23:19 random
drwxr-xr-x 4 root root 0 Nov 14 23:19 raw_syscalls
drwxr-xr-x 3 root root 0 Nov 14 23:19 rcu
drwxr-xr-x 6 root root 0 Nov 14 23:19 rpm
drwxr-xr-x 20 root root 0 Nov 14 23:19 sched
drwxr-xr-x 7 root root 0 Nov 14 23:19 scsi
drwxr-xr-x 4 root root 0 Nov 14 23:19 signal
drwxr-xr-x 5 root root 0 Nov 14 23:19 skb
drwxr-xr-x 4 root root 0 Nov 14 23:19 sock
drwxr-xr-x 10 root root 0 Nov 14 23:19 sunrpc
drwxr-xr-x 538 root root 0 Nov 14 23:19 syscalls
drwxr-xr-x 4 root root 0 Nov 14 23:19 task

```

|            |    |      |      |   |              |           |
|------------|----|------|------|---|--------------|-----------|
| drwxr-xr-x | 14 | root | root | 0 | Nov 14 23:19 | timer     |
| drwxr-xr-x | 3  | root | root | 0 | Nov 14 23:19 | udp       |
| drwxr-xr-x | 21 | root | root | 0 | Nov 14 23:19 | vmscan    |
| drwxr-xr-x | 3  | root | root | 0 | Nov 14 23:19 | vsyscall  |
| drwxr-xr-x | 6  | root | root | 0 | Nov 14 23:19 | workqueue |
| drwxr-xr-x | 26 | root | root | 0 | Nov 14 23:19 | writeback |

Each one of these subdirectories corresponds to a 'subsystem' and contains yet again more subdirectories, each one of those finally corresponding to a tracepoint. For example, here are the contents of the 'kmem' subsystem:

```
root@sugarbay:/sys/kernel/debug/tracing/events# cd kmem
root@sugarbay:/sys/kernel/debug/tracing/events/kmem# ls -al
drwxr-xr-x 14 root root 0 Nov 14 23:19 .
drwxr-xr-x 38 root root 0 Nov 14 23:19 ..
-rw-r--r-- 1 root root 0 Nov 14 23:19 enable
-rw-r--r-- 1 root root 0 Nov 14 23:19 filter
drwxr-xr-x 2 root root 0 Nov 14 23:19 kfree
drwxr-xr-x 2 root root 0 Nov 14 23:19 kmalloc
drwxr-xr-x 2 root root 0 Nov 14 23:19 kmalloc_node
drwxr-xr-x 2 root root 0 Nov 14 23:19 kmem_cache_alloc
drwxr-xr-x 2 root root 0 Nov 14 23:19 kmem_cache_alloc_node
drwxr-xr-x 2 root root 0 Nov 14 23:19 kmem_cache_free
drwxr-xr-x 2 root root 0 Nov 14 23:19 mm_page_alloc
drwxr-xr-x 2 root root 0 Nov 14 23:19 mm_page_alloc_extfrag
drwxr-xr-x 2 root root 0 Nov 14 23:19 mm_page_alloc_zone_locked
drwxr-xr-x 2 root root 0 Nov 14 23:19 mm_page_free
drwxr-xr-x 2 root root 0 Nov 14 23:19 mm_page_free_batched
drwxr-xr-x 2 root root 0 Nov 14 23:19 mm_page_pcpu_drain
```

Let's see what's inside the subdirectory for a specific tracepoint, in this case the one for `kmallocc`:

```
root@sugarbay:/sys/kernel/debug/tracing/events/kmem# cd kmallo
root@sugarbay:/sys/kernel/debug/tracing/events/kmem/kmallo# ls -al
drwxr-xr-x 2 root root 0 Nov 14 23:19 .
drwxr-xr-x 14 root root 0 Nov 14 23:19 ..
-rw-r--r-- 1 root root 0 Nov 14 23:19 enable
-rw-r--r-- 1 root root 0 Nov 14 23:19 filter
-r--r--r-- 1 root root 0 Nov 14 23:19 format
-r--r--r-- 1 root root 0 Nov 14 23:19 id
```

The 'format' file for the tracepoint describes the event in memory, which is used by the various tracing tools that now make use of these tracepoint to parse the event and make sense of it, along with a 'print fmt' field that allows tools like ftrace to display the event as text. Here's what the format of the kmalloc event looks like:

```

root@sugarbay:/sys/kernel/debug/tracing/events/kmem/kmalloc# cat format
name: kmalloc
ID: 313
format:
 field:unsigned short common_type; offset:0; size:2; signed:0;
 field:unsigned char common_flags; offset:2; size:1; signed:0;
 field:unsigned char common_preempt_count; offset:3; size:1; signed:0;
 field:int common_pid; offset:4; size:4; signed:1;
 field:int common_padding; offset:8; size:4; signed:1;

 field:unsigned long call_site; offset:16; size:8; signed:0;
 field:const void * ptr; offset:24; size:8; signed:0;
 field:size_t bytes_req; offset:32; size:8; signed:0;
 field:size_t bytes_alloc; offset:40; size:8; signed:0;
 field:gfp_t gfp_flags; offset:48; size:4; signed:0;

print fmt: "call_site=%lx ptr=%p bytes_req=%zu bytes_alloc=%zu gfp_flags=%s", REC->call_site, REC->ptr, REC->bytes_req, RI
(REC->gfp_flags) ? _print_flags(REC->gfp_flags, "|", {(unsigned long) (((gfp_t)0x10u) | ((gfp_t)0x40u) | ((gfp_t)0x80
gfp_t)0x20000u) | ((gfp_t)0x02u) | ((gfp_t)0x08u)) | ((gfp_t)0x4000u) | ((gfp_t)0x10000u) | ((gfp_t)0x1000u) | ((gfp
gfp_t)0x4000000u)), "GFP_TRANSHUGE"}, {(unsigned long) (((gfp_t)0x10u) | ((gfp_t)0x40u) | ((gfp_t)0x80u) | ((gfp_t)0x200
gfp_t)0x02u) | ((gfp_t)0x08u)), "GFP_HIGHUSER_MOVABLE"}, {(unsigned long) (((gfp_t)0x10u) | ((gfp_t)0x40u) | ((gfp_t)0
gfp_t)0x20000u) | ((gfp_t)0x02u)), "GFP_HIGHUSER"}, {(unsigned long) (((gfp_t)0x10u) | ((gfp_t)0x40u) | ((gfp_t)0x80u)
gfp_t)0x200000u), "GFP_USER"}, {(unsigned long) (((gfp_t)0x10u) | ((gfp_t)0x40u) | ((gfp_t)0x80u) | ((gfp_t)0x800000u)
{(unsigned long) (((gfp_t)0x10u) | ((gfp_t)0x40u) | ((gfp_t)0x80u)), "GFP_KERNEL"}, {(unsigned long) (((gfp_t)0x10u) |
"GFP_NOFS"}, {(unsigned long) (((gfp_t)0x20u)), "GFP_ATOMIC"}, {(unsigned long) (((gfp_t)0x10u)), "GFP_NOIO"}, {(unsigned
gfp_t)0x20u), "GFP_HIGH"}, {(unsigned long) ((gfp_t)0x10u), "GFP_WAIT"}, {(unsigned long) ((gfp_t)0x40u), "GFP_IO"}, {(un
gfp_t)0x100u), "GFP_COLD"}, {(unsigned long) ((gfp_t)0x200u), "GFP_NOWARN"}, {(unsigned long) ((gfp_t)0x400u), "GFP_REPEA
long) ((gfp_t)0x800u), "GFP_NOFAIL"}, {(unsigned long) ((gfp_t)0x1000u), "GFP_NORETRY"}, {(unsigned long) ((gfp_t)0x0
{(unsigned long) ((gfp_t)0x8000u), "GFP_ZERO"}, {(unsigned long) ((gfp_t)0x10000u), "GFP_NOMEMALLOC"}, {(unsigned long) ((
"GFP_HARDWALL"}, {(unsigned long) ((gfp_t)0x400000u), "GFP_THISNODE"}, {(unsigned long) ((gfp_t)0x800000u), "GFP_RECLAIMAB
long) ((gfp_t)0x08u), "GFP_MOVABLE"}, {(unsigned long) ((gfp_t)0), "GFP_NOTRACK"}, {(unsigned long) ((gfp_t)0x4000000u), "
{(unsigned long) ((gfp_t)0x8000000u), "GFP_OTHER_NODE"} : "GFP_NOWAIT"

```

The 'enable' file in the tracepoint directory is what allows the user (or tools such as trace-cmd) to actually turn the tracepoint on and off. When enabled, the corresponding tracepoint will start appearing in the ftrace 'trace' file described previously. For example, this turns on the kmallocc tracepoint:

```
root@sugarbay:/sys/kernel/debug/tracing/events/kmem/kmalloc# echo 1 > enable
```

At the moment, we're not interested in the function tracer or some other tracer that might be in effect, so we first turn it off, but if we do that, we still need to turn tracing on in order to see the events in the output buffer:

```
root@sugarbay:/sys/kernel/debug/tracing# echo nop > current_tracer
root@sugarbay:/sys/kernel/debug/tracing# echo 1 > tracing on
```

Now, if we look at the the 'trace' file, we see nothing but the kmalloc events we just turned on:

```
root@sugarbay:/sys/kernel/debug/tracing# cat trace | less
tracer: nop
#
entries-in-buffer/entries-written: 1897/1897 #P:8
```

```

#
_-----> irqsoft
/-----> need-resched
/ /-----> hardirq/softirq
|| /-----> preempt-depth
||| /-----> delay
#
TASK-PID CPU# ||||| TIMESTAMP FUNCTION
#
dropbear-1465 [000] ...1 18154.620753: kmalloc: call_site=ffffffff816650d4 ptr=ffff8800729c3000 bytes_req=2048 by
<idle>-0 [000] ...3 18154.621640: kmalloc: call_site=ffffffff81619b36 ptr=ffff88006d555800 bytes_req=512 byt
<idle>-0 [000] ...3 18154.621656: kmalloc: call_site=ffffffff81619b36 ptr=ffff88006d555800 bytes_req=512 byt
matchbox-termin-1361 [001] ...1 18154.755472: kmalloc: call_site=ffffffff81614050 ptr=ffff88006d5f0e00 bytes_req=512 byt
Xorg-1264 [002] ...1 18154.755581: kmalloc: call_site=ffffffff8141abe8 ptr=ffff8800734f4cc0 bytes_req=168 byt
Xorg-1264 [002] ...1 18154.755583: kmalloc: call_site=ffffffff814192a3 ptr=ffff88001f822520 bytes_req=24 byte:
Xorg-1264 [002] ...1 18154.755589: kmalloc: call_site=ffffffff81419edb ptr=ffff8800721a2f00 bytes_req=64 byte:
matchbox-termin-1361 [001] ...1 18155.354594: kmalloc: call_site=ffffffff81614050 ptr=ffff88006d555800 bytes_req=576 byt
Xorg-1264 [002] ...1 18155.354703: kmalloc: call_site=ffffffff8141abe8 ptr=ffff8800734f4cc0 bytes_req=168 byt
Xorg-1264 [002] ...1 18155.354705: kmalloc: call_site=ffffffff814192a3 ptr=ffff88001f822520 bytes_req=24 byte:
Xorg-1264 [002] ...1 18155.354711: kmalloc: call_site=ffffffff81419edb ptr=ffff8800721a2f00 bytes_req=64 byte:
dropbear-1465 [000] ...1 18155.673319: kmalloc: call_site=ffffffff81619b36 ptr=ffff88006d555800 bytes_req=512 byt
dropbear-1465 [000] ...1 18155.673525: kmalloc: call_site=ffffffff816650d4 ptr=ffff8800729c3000 bytes_req=2048 by
<idle>-0 [000] ...3 18155.674821: kmalloc: call_site=ffffffff81619b36 ptr=ffff88006d554800 bytes_req=512 byt
<idle>-0 [000] ...3 18155.793014: kmalloc: call_site=ffffffff81619b36 ptr=ffff88006d554800 bytes_req=512 byt
dropbear-1465 [000] ...1 18155.793219: kmalloc: call_site=ffffffff816650d4 ptr=ffff8800729c3000 bytes_req=2048 by
<idle>-0 [000] ...3 18155.794147: kmalloc: call_site=ffffffff81619b36 ptr=ffff88006d555800 bytes_req=512 byt
<idle>-0 [000] ...3 18155.936705: kmalloc: call_site=ffffffff81619b36 ptr=ffff88006d555800 bytes_req=512 byt
dropbear-1465 [000] ...1 18155.936910: kmalloc: call_site=ffffffff816650d4 ptr=ffff8800729c3000 bytes_req=2048 by
<idle>-0 [000] ...3 18155.937869: kmalloc: call_site=ffffffff81619b36 ptr=ffff88006d554800 bytes_req=512 byt
matchbox-termin-1361 [001] ...1 18155.953667: kmalloc: call_site=ffffffff81614050 ptr=ffff88006d5f2000 bytes_req=512 byt
Xorg-1264 [002] ...1 18155.953775: kmalloc: call_site=ffffffff8141abe8 ptr=ffff8800734f4cc0 bytes_req=168 byt
Xorg-1264 [002] ...1 18155.953777: kmalloc: call_site=ffffffff814192a3 ptr=ffff88001f822520 bytes_req=24 byte:
Xorg-1264 [002] ...1 18155.953783: kmalloc: call_site=ffffffff81419edb ptr=ffff8800721a2f00 bytes_req=64 byte:
<idle>-0 [000] ...3 18156.176053: kmalloc: call_site=ffffffff81619b36 ptr=ffff88006d554800 bytes_req=512 byt
dropbear-1465 [000] ...1 18156.176257: kmalloc: call_site=ffffffff816650d4 ptr=ffff8800729c3000 bytes_req=2048 by
<idle>-0 [000] ...3 18156.177717: kmalloc: call_site=ffffffff81619b36 ptr=ffff88006d555800 bytes_req=512 byt
<idle>-0 [000] ...3 18156.399229: kmalloc: call_site=ffffffff81619b36 ptr=ffff88006d555800 bytes_req=512 byt
dropbear-1465 [000] ...1 18156.399434: kmalloc: call_site=ffffffff816650d4 ptr=ffff8800729c3000 bytes_req=2048 by
<idle>-0 [000] ...3 18156.400660: kmalloc: call_site=ffffffff81619b36 ptr=ffff88006d554800 bytes_req=512 byt
matchbox-termin-1361 [001] ...1 18156.552800: kmalloc: call_site=ffffffff81614050 ptr=ffff88006d534800 bytes_req=576 byt

```

To again disable the kmalloc event, we need to send 0 to the enable file:

```
root@sugarbay:/sys/kernel/debug/tracing/events/kmem/kmalloc# echo 0 > enable
```

You can enable any number of events or complete subsystems (by using the 'enable' file in the subsystem directory) and get an arbitrarily fine-grained idea of what's going on in the system by enabling as many of the appropriate tracepoints as applicable.

A number of the tools described in this HOWTO do just that, including trace-cmd and kernelshark in the next section.

**Tying it Together:** These tracepoints and their representation are used not only by ftrace, but by many of the other tools covered in this document and they form a central point of integration for the various tracers available in Linux. They form a central part of the instrumentation for the following tools: perf, ltng, ftrace, blktrace and SystemTap

**Tying it Together:** Eventually all the special-purpose tracers currently available in /sys/kernel/debug/tracing will be removed and replaced with equivalent tracers based on the 'trace events' subsystem.

#### 19.2.4. trace-cmd/kernelshark

trace-cmd is essentially an extensive command-line 'wrapper' interface that hides the details of all the individual files in /sys/kernel/debug/tracing, allowing users to specify specific particular events within the /sys/kernel/debug/tracing/events/ subdirectory and to collect traces and avoid having to deal with those details directly.

As yet another layer on top of that, kernelshark provides a GUI that allows users to start and stop traces and specify sets of events using an intuitive interface, and view the output as both trace events and as a per-CPU graphical display. It directly uses 'trace-cmd' as the plumbing that accomplishes all that underneath the covers (and actually displays the trace-cmd command it uses, as we'll see).

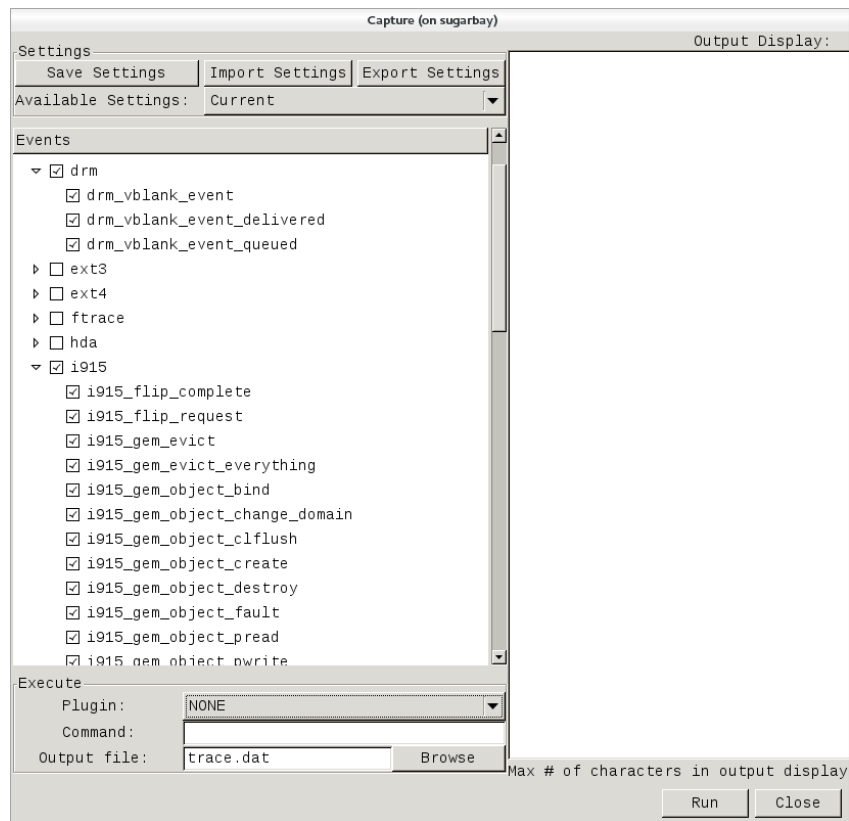
To start a trace using kernelshark, first start kernelshark:

```
root@sugarbay:~# kernelshark
```

Then bring up the 'Capture' dialog by choosing from the kernelshark menu:

```
Capture | Record
```

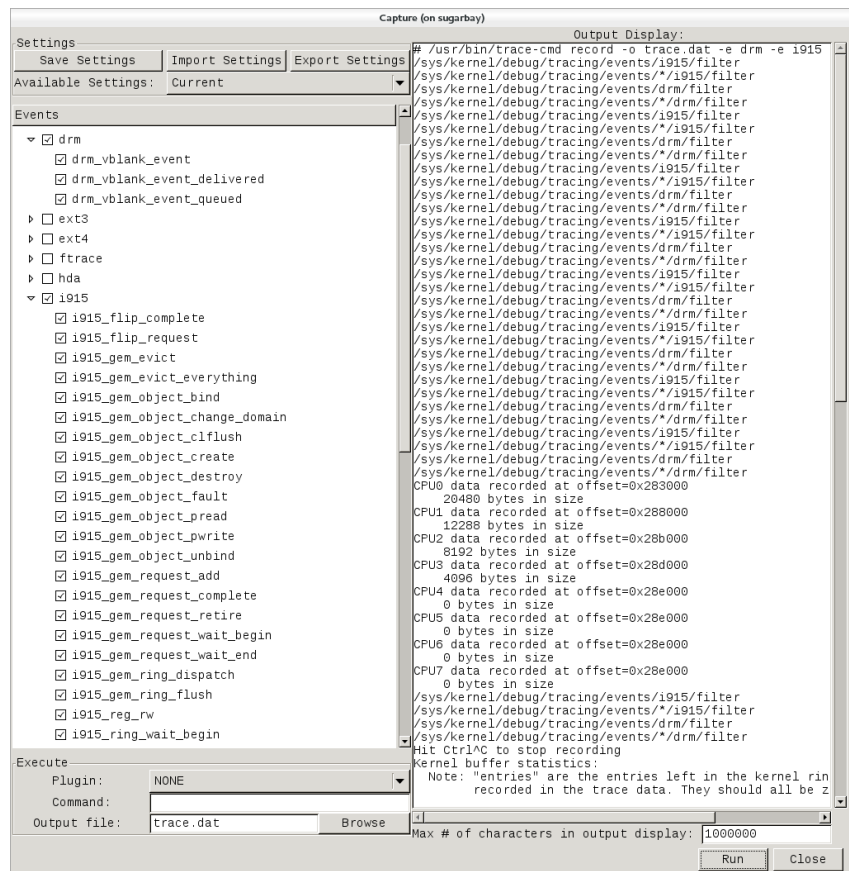
That will display the following dialog, which allows you to choose one or more events (or even one or more complete subsystems) to trace:



Note that these are exactly the same sets of events described in the previous trace events subsystem section, and in fact is where trace-cmd gets them for kernelshark.

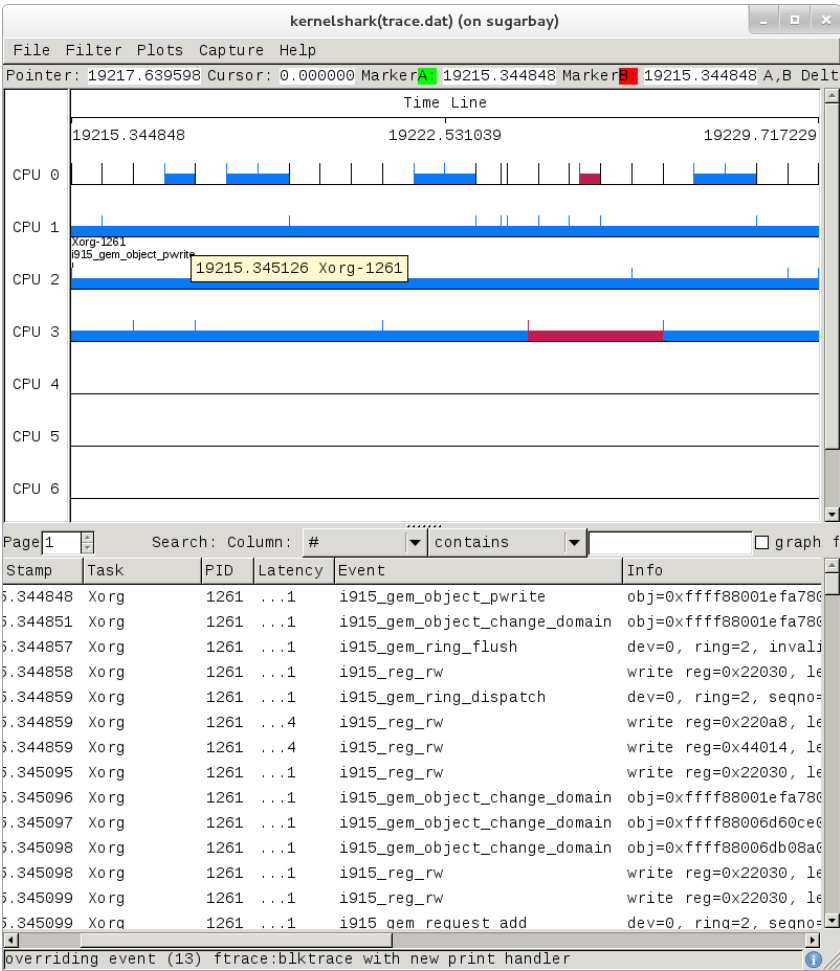
In the above screenshot, we've decided to explore the graphics subsystem a bit and so have chosen to trace all the tracepoints contained within the 'i915' and 'drm' subsystems.

After doing that, we can start and stop the trace using the 'Run' and 'Stop' button on the lower right corner of the dialog (the same button will turn into the 'Stop' button after the trace has started):

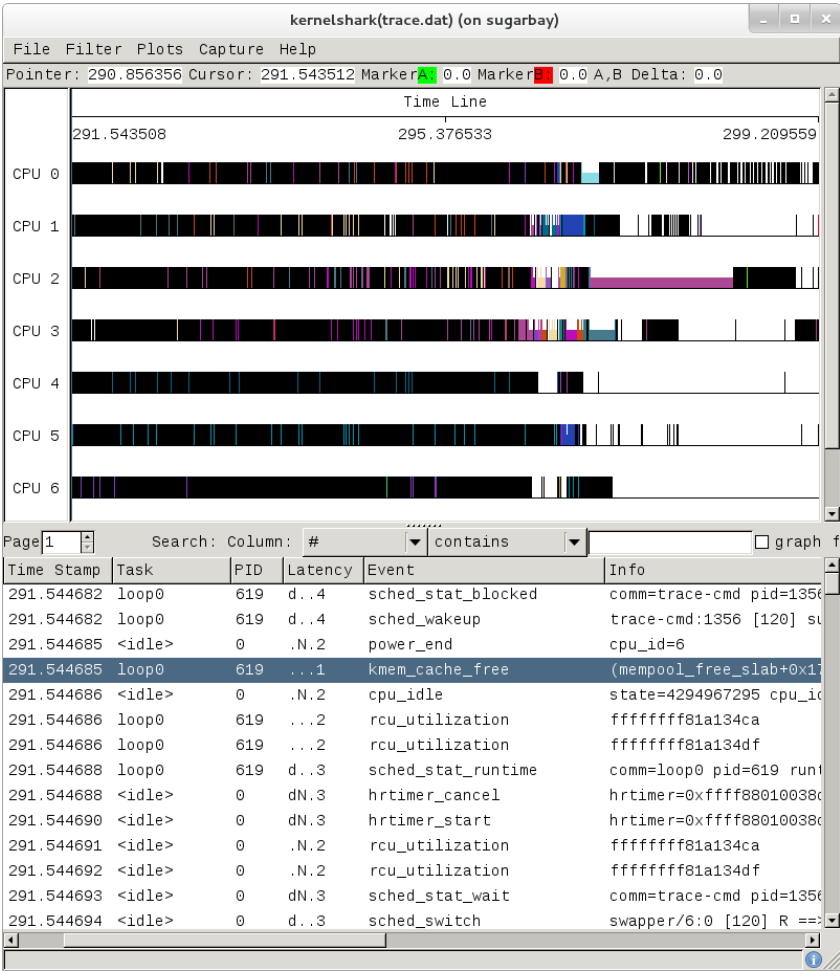


Notice that the right-hand pane shows the exact trace-cmd command-line that's used to run the trace, along with the results of the trace-cmd run.

Once the 'Stop' button is pressed, the graphical view magically fills up with a colorful per-cpu display of the trace data, along with the detailed event listing below that:



Here's another example, this time a display resulting from tracing 'all events':



The tool is pretty self-explanatory, but for more detailed information on navigating through the data, see the [kernelshark website](https://www.kernelshark.org/).

### 19.2.5. Documentation¶

The documentation for ftrace can be found in the kernel Documentation directory:

```
Documentation/trace/ftrace.txt
```

The documentation for the trace event subsystem can also be found in the kernel Documentation directory:

```
Documentation/trace/events.txt
```

There is a nice series of articles on using ftrace and trace-cmd at LWN:

- [Debugging the kernel using Ftrace - part 1](#)
- [Debugging the kernel using Ftrace - part 2](#)
- [Secrets of the Ftrace function tracer](#)
- [trace-cmd: A front-end for Ftrace](#)

There's more detailed documentation kernelshark usage here: [KernelShark](#)

An amusing yet useful README (a tracing mini-HOWTO) can be found in /sys/kernel/debug/tracing/README.

### 19.3. systemtap¶

SystemTap is a system-wide script-based tracing and profiling tool.

SystemTap scripts are C-like programs that are executed in the kernel to gather/print/aggregate data extracted from the context they end up being invoked under.

For example, this probe from the [SystemTap tutorial](#) simply prints a line every time any process on the system open()s a file. For each line, it prints the executable name of the program that opened the file, along with its PID, and the name of the file it opened (or tried to open), which it extracts from the open syscall's argstr.

```
probe syscall.open
{
 printf ("%s(%d) open (%s)\n", execname(), pid(), argstr)
}

probe timer.ms(4000) # after 4 seconds
{
 exit ()
}
```

Normally, to execute this probe, you'd simply install systemtap on the system you want to probe, and directly run the probe on that system e.g. assuming the name of the file containing the above text is trace\_open.stp:

```
stap trace_open.stp
```

What systemtap does under the covers to run this probe is 1) parse and convert the probe to an equivalent 'C' form, 2) compile the 'C' form into a kernel module, 3) insert the module into the kernel, which arms it, and 4) collect the data generated by the probe and display it to the user.

In order to accomplish steps 1 and 2, the 'stap' program needs access to the kernel build system that produced the kernel that the probed system is running. In the case of a typical embedded system (the 'target'), the kernel build system unfortunately isn't typically part of the image running on the target. It is normally available on the 'host' system that produced the target image however; in such cases, steps 1 and 2 are executed on the host system, and steps 3 and 4 are executed on the target system, using only the systemtap 'runtime'.

The systemtap support in Yocto assumes that only steps 3 and 4 are run on the target; it is possible to do everything on the target, but this section assumes only the typical embedded use-case.

So basically what you need to do in order to run a systemtap script on the target is to 1) on the host system, compile the probe into a kernel module that makes sense to the target, 2) copy the module onto the target system and 3) insert the module into the target kernel, which arms it, and 4) collect the data generated by the probe and display it to the user.

#### 19.3.1. Setup¶

Those are a lot of steps and a lot of details, but fortunately Yocto includes a script called 'crosstap' that will take care of those details, allowing you to simply execute a systemtap script on the remote target, with arguments if necessary.

In order to do this from a remote host, however, you need to have access to the build for the image you booted. The 'crosstap' script provides details on how to do this if you run the script on the host without having done a build:

#### Note

SystemTap, which uses 'crosstap', assumes you can establish an ssh connection to the remote target. Please refer to the crosstap wiki page for details on verifying ssh connections at [https://wiki.yoctoproject.org/wiki/Tracing\\_and\\_Profiling#systemtap](https://wiki.yoctoproject.org/wiki/Tracing_and_Profiling#systemtap). Also, the ability to ssh into the target system is not enabled by default in \*-minimal images.

```
$ crosstap root@192.168.1.88 trace_open.stp
```

```
Error: No target kernel build found.
Did you forget to create a local build of your image?
```

'crosstap' requires a local sdk build of the target system (or a build that includes 'tools-profile') in order to build kernel modules that can probe the target system.

Practically speaking, that means you need to do the following:

- If you're running a pre-built image, download the release and/or BSP tarballs used to build the image.
- If you're working from git sources, just clone the metadata and BSP layers needed to build the image you'll be booting.
- Make sure you're properly set up to build a new image (see the BSP README and/or the widely available basic documentation that discusses how to build images).
- Build an -sdk version of the image e.g.:  
`$ bitbake core-image-sato-sdk`
- OR
- Build a non-sdk image but include the profiling tools:  
`[ edit local.conf and add 'tools-profile' to the end of the EXTRA_IMAGE_FEATURES variable ]`  
`$ bitbake core-image-sato`

Once you've build the image on the host system, you're ready to boot it (or the equivalent pre-built image) and use 'crosstap' to probe it (you need to source the environment as usual first):

```
$ source oe-init-build-env
$ cd ~/my/systemtap/scripts
$ crosstap root@192.168.1.xxx myscript.stp
```

So essentially what you need to do is build an SDK image or image with 'tools-profile' as detailed in the "[General Setup](#)" section of this manual, and boot the resulting target image.

## Note

If you have a build directory containing multiple machines, you need to have the MACHINE you're connecting to selected in local.conf, and the kernel in that machine's build directory must match the kernel on the booted system exactly, or you'll get the above 'crosstap' message when you try to invoke a script.

### 19.3.2. Running a Script on a Target¶

Once you've done that, you should be able to run a systemtap script on the target:

```
$ cd /path/to/yocto
$ source oe-init-build-env

Shell environment set up for builds.

You can now run 'bitbake <target>'

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

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

Once you've done that, you can cd to whatever directory contains your scripts and use 'crosstap' to run the script:

```
$ cd /path/to/my/systemtap/script
$ crosstap root@192.168.7.2 trace_open.stp
```

If you get an error connecting to the target e.g.:

```
$ crosstap root@192.168.7.2 trace_open.stp
error establishing ssh connection on remote 'root@192.168.7.2'
```

Try ssh'ing to the target and see what happens:

```
$ ssh root@192.168.7.2
```

A lot of the time, connection problems are due specifying a wrong IP address or having a 'host key verification error'.

If everything worked as planned, you should see something like this (enter the password when prompted, or press enter if it's set up to use no password):

```
$ crosstap root@192.168.7.2 trace_open.stp
root@192.168.7.2's password:
matchbox-termin(1036) open ("/tmp/vte3FS2LW", O_RDWR|O_CREAT|O_EXCL|O_LARGEFILE, 0600)
matchbox-termin(1036) open ("/tmp/vteJMC7LW", O_RDWR|O_CREAT|O_EXCL|O_LARGEFILE, 0600)
```

### 19.3.3. Documentation¶

The SystemTap language reference can be found here: [SystemTap Language Reference](#)

Links to other SystemTap documents, tutorials, and examples can be found here: [SystemTap documentation page](#)

## 19.4. Sysprof¶



Sysprof is a very easy to use system-wide profiler that consists of a single window with three panes and a few buttons which allow you to start, stop, and view the profile from one place.

### 19.4.1. Setup

For this section, we'll assume you've already performed the basic setup outlined in the General Setup section.

Sysprof is a GUI-based application that runs on the target system. For the rest of this document we assume you've ssh'ed to the host and will be running Sysprof on the target (you can use the '-X' option to ssh and have the Sysprof GUI run on the target but display remotely on the host if you want).

### 19.4.2. Basic Usage

To start profiling the system, you simply press the 'Start' button. To stop profiling and to start viewing the profile data in one easy step, press the 'Profile' button.

Once you've pressed the profile button, the three panes will fill up with profiling data:

| Functions                   | Self    | Total    |
|-----------------------------|---------|----------|
| [Everything]                | 0.00 %  | 100.00 % |
| - - kernel - -              | 0.00 %  | 88.71 %  |
| [wget]                      | 0.00 %  | 79.94 %  |
| __do_softirq                | 2.19 %  | 41.68 %  |
| syscalls_call               | 0.05 %  | 39.88 %  |
| _raw_spin_unlock_irqrestore | 38.36 % | 38.83 %  |
| run_timer_softirq           | 0.33 %  | 36.87 %  |
| __queue_work                | 0.15 %  | 34.92 %  |
| delayed_work_timer_fn       | 0.08 %  | 34.92 %  |
| __write_nocancel            | 0.13 %  | 30.83 %  |
| __read_nocancel             | 0.67 %  | 21.88 %  |
| __poll_nocancel             | 0.49 %  | 20.75 %  |
| sys_read                    | 0.46 %  | 13.11 %  |
| sys_write                   | 0.03 %  | 12.99 %  |
| vfs_write                   | 0.03 %  | 12.86 %  |
| svs_noll                    | 0.10 %  | 12.57 %  |
| syscalls_call               | 0.21 %  | 12.86 %  |
| - - kernel - -              | 0.26 %  | 0.26 %   |

| Callers        | Self   | Total   |
|----------------|--------|---------|
| syscalls_call  | 0.21 % | 12.86 % |
| - - kernel - - | 0.26 % | 0.26 %  |

| Descendants              | Self   | Cumulative |
|--------------------------|--------|------------|
| sys_read                 | 0.46 % | 13.11 %    |
| vfs_read                 | 0.46 % | 12.09 %    |
| do_sync_read             | 0.23 % | 10.49 %    |
| sock_aio_read            | 0.39 % | 9.90 %     |
| inet_recvmmsg            | 0.13 % | 8.92 %     |
| tcp_recvmmsg             | 1.05 % | 8.23 %     |
| skb_copy_datagram_iov... | 0.41 % | 2.98 %     |
| memcpy_toiovec           | 0.28 % | 2.47 %     |
| copy_to_user             | 0.13 % | 2.16 %     |
| __copy_to_user_ll        | 2.03 % | 2.03 %     |
| copy_to_user             | 0.10 % | 0.10 %     |
| _kfree_skb               | 0.03 % | 1.83 %     |
| lock_sock_nested         | 0.15 % | 0.93 %     |
| release_sock             | 0.13 % | 0.82 %     |
| tcp_cleanup_rbuf         | 0.18 % | 0.18 %     |
| tcp_rcv_space_adjust     | 0.10 % | 0.10 %     |
| memcpy_toiovec           | 0.08 % | 0.08 %     |
| local_bh_enable          | 0.08 % | 0.08 %     |
| skb_release_data         | 0.05 % | 0.05 %     |
| kmem_cache_free          | 0.05 % | 0.05 %     |
| _raw_spin_lock_bh        | 0.03 % | 0.03 %     |
| skb_release_head_state   | 0.03 % | 0.03 %     |
| _raw_spin_unlock         | 0.03 % | 0.03 %     |
| tcp_rcv_space_adjust     | 0.13 % | 0.13 %     |
| tcp_cleanup_rbuf         | 0.10 % | 0.10 %     |
| skb_copy_datagram_iovec  | 0.10 % | 0.10 %     |
| _kfree_skb               | 0.08 % | 0.08 %     |

The left pane shows a list of functions and processes. Selecting one of those expands that function in the right pane, showing all its callees. Note that this caller-oriented display is essentially the inverse of perf's default callee-oriented callchain display.

In the screenshot above, we're focusing on `__copy_to_user_ll()` and looking up the callchain we can see that one of the callers of `__copy_to_user_ll` is `sys_read()` and the complete callpath between them. Notice that this is essentially a portion of the same information we saw in the perf display shown in the perf section of this page.

| Functions                   | Self    | Total    |
|-----------------------------|---------|----------|
| [Everything]                | 0.00 %  | 100.00 % |
| - - kernel - -              | 0.00 %  | 88.71 %  |
| [wget]                      | 0.00 %  | 79.94 %  |
| __do_softirq                | 2.19 %  | 41.68 %  |
| syscalls_call               | 0.05 %  | 39.88 %  |
| _raw_spin_unlock_irqrestore | 38.36 % | 38.83 %  |
| run_timer_softirq           | 0.33 %  | 36.87 %  |
| __queue_work                | 0.15 %  | 34.92 %  |
| delayed_work_timer_fn       | 0.08 %  | 34.92 %  |
| __write_nocancel            | 0.13 %  | 30.83 %  |
| __read_nocancel             | 0.67 %  | 21.88 %  |
| __poll_nocancel             | 0.49 %  | 20.75 %  |
| sys_read                    | 0.46 %  | 13.11 %  |
| sys_write                   | 0.03 %  | 12.99 %  |
| vfs_write                   | 0.03 %  | 12.86 %  |
| svs_noll                    | 0.10 %  | 12.57 %  |
| syscalls_call               | 0.03 %  | 12.99 %  |

| Callers       | Self   | Total   |
|---------------|--------|---------|
| syscalls_call | 0.03 % | 12.99 % |

| Descendants                      | Self   | Cumulative |
|----------------------------------|--------|------------|
| sys_write                        | 0.03 % | 12.99 %    |
| vfs_write                        | 0.03 % | 12.86 %    |
| do_sync_write                    | 0.21 % | 12.50 %    |
| ext4_file_write                  | 0.18 % | 11.98 %    |
| generic_file_aio_write           | 0.03 % | 11.67 %    |
| __generic_file_aio_write         | 0.15 % | 11.44 %    |
| generic_file_buffered_write      | 0.08 % | 9.13 %     |
| ext4_da_write_end                | 0.33 % | 4.27 %     |
| ext4_da_write_begin              | 0.08 % | 3.09 %     |
| iov_iter_copy_from_user_atomic   | 0.05 % | 1.03 %     |
| __copy_from_user_ll_nofzero      | 0.72 % | 0.72 %     |
| kmap_atomic                      | 0.03 % | 0.15 %     |
| kunmap_atomic                    | 0.08 % | 0.10 %     |
| balance_dirty_pages_ratelimit... | 0.08 % | 0.18 %     |
| __copy_from_user_ll_nofzero      | 0.10 % | 0.10 %     |
| add_preempt_count                | 0.05 % | 0.05 %     |
| kunmap_atomic                    | 0.05 % | 0.05 %     |
| kmap_atomic                      | 0.05 % | 0.05 %     |
| sub_preempt_count                | 0.05 % | 0.05 %     |
| mark_page_accessed               | 0.03 % | 0.05 %     |
| iov_iter_advance                 | 0.03 % | 0.03 %     |
| grab_cache_page_write_begin      | 0.03 % | 0.03 %     |
| generic_write_end                | 0.03 % | 0.03 %     |
| __block_write_begin              | 0.03 % | 0.03 %     |
| preempt_schedule                 | 0.00 % | 0.03 %     |
| file_update_time                 | 0.10 % | 1.85 %     |
| mark_page_accessed               | 0.05 % | 0.05 %     |
| ext4_da_write_end                | 0.05 % | 0.05 %     |

Similarly, the above is a snapshot of the Sysprof display of a copy-from-user callchain.

Finally, looking at the third Sysprof pane in the lower left, we can see a list of all the callers of a particular function selected in the top left pane. In this case, the lower pane is showing all the callers of `__mark_inode_dirty`:

| Functions                      | Self          | Total         |
|--------------------------------|---------------|---------------|
| sock_recv_read                 | 0.41 %        | 9.90 %        |
| [sysprof]                      | 0.00 %        | 9.49 %        |
| inet_recvmsg                   | 0.31 %        | 9.13 %        |
| generic_file_buffered_write    | 0.08 %        | 9.13 %        |
| tcp_recvmsg                    | 1.26 %        | 8.46 %        |
| _raw_spin_unlock_irq           | 7.07 %        | 7.46 %        |
| poll_schedule_timeout          | 0.15 %        | 7.20 %        |
| schedule_hrtimeout_range       | 0.08 %        | 7.05 %        |
| schedule_hrtimeout_range_clock | 0.23 %        | 6.92 %        |
| _schedule                      | 0.33 %        | 6.04 %        |
| schedule                       | 0.10 %        | 5.63 %        |
| kthread                        | 0.00 %        | 5.37 %        |
| kernel_thread_helper           | 0.00 %        | 5.37 %        |
| finish_task_switch             | 0.10 %        | 5.27 %        |
| <b>__mark_inode_dirty</b>      | <b>0.08 %</b> | <b>4.40 %</b> |
| ext4_da_write_end              | 0.39 %        | 4.32 %        |

| Callers                      | Self   | Total  |
|------------------------------|--------|--------|
| generic_write_end            | 0.00 % | 2.88 % |
| file_update_time             | 0.03 % | 1.44 % |
| ext4_da_update_reserve_space | 0.00 % | 0.03 % |
| _generic_file_aio_write      | 0.03 % | 0.03 % |
| ext4_da_write_end            | 0.03 % | 0.03 % |

| Descendants                   | Self   | Cumulative |
|-------------------------------|--------|------------|
| __mark_inode_dirty            | 0.08 % | 4.40 %     |
| ext4_dirty_inode              | 0.08 % | 4.06 %     |
| ext4_mark_inode_dirty         | 0.08 % | 3.11 %     |
| ext4_reserve_inode_write      | 0.10 % | 2.34 %     |
| ext4_journal_get_write_access | 0.03 % | 1.16 %     |
| ext4_get_inode_loc            | 0.00 % | 0.95 %     |
| jbd2_journal_get_write_access | 0.08 % | 0.08 %     |
| ext4_get_inode_loc            | 0.05 % | 0.05 %     |
| ext4_mark_inode_dirty         | 0.36 % | 0.62 %     |
| ext4_handle_dirty_metadata    | 0.03 % | 0.03 %     |
| ext4_get_inode_flags          | 0.03 % | 0.03 %     |
| ext4_journal_get_write_access | 0.03 % | 0.03 %     |
| ext4_journal_start_sb         | 0.08 % | 0.46 %     |
| __ext4_journal_stop           | 0.05 % | 0.18 %     |
| jbd2_journal_stop             | 0.08 % | 0.08 %     |
| ext4_reserve_inode_write      | 0.05 % | 0.05 %     |
| ext4_mark_inode_dirty         | 0.05 % | 0.05 %     |
| jbd2_journal_start            | 0.05 % | 0.05 %     |
| __ext4_journal_stop           | 0.10 % | 0.10 %     |
| ext4_mark_inode_dirty         | 0.08 % | 0.08 %     |
| ext4_journal_start_sb         | 0.08 % | 0.08 %     |

Double-clicking on one of those functions will in turn change the focus to the selected function, and so on.

**Tying it Together:** If you like sysprof's 'caller-oriented' display, you may be able to approximate it in other tools as well. For example, 'perf report' has the -g (--call-graph) option that you can experiment with; one of the options is 'caller' for an inverted caller-based callgraph display.

### 19.4.3. Documentation¶

There doesn't seem to be any documentation for Sysprof, but maybe that's because it's pretty self-explanatory. The Sysprof website, however, is here: [Sysprof, System-wide Performance Profiler for Linux](#)

## 19.5. LTTng (Linux Trace Toolkit, next generation)¶

### 19.5.1. Setup¶

For this section, we'll assume you've already performed the basic setup outlined in the General Setup section. LTTng is run on the target system by ssh'ing to it.

### 19.5.2. Collecting and Viewing Traces¶

Once you've applied the above commits and built and booted your image (you need to build the core-image-sato-sdk image or use one of the other methods described in the General Setup section), you're ready to start tracing.

#### 19.5.2.1. Collecting and viewing a trace on the target (inside a shell)¶

First, from the host, ssh to the target:

```
$ ssh -l root 192.168.1.47
The authenticity of host '192.168.1.47 (192.168.1.47)' can't be established.
RSA key fingerprint is 23:bd:c8:b1:a8:71:52:00:ee:00:4f:64:9e:10:b9:7e.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.1.47' (RSA) to the list of known hosts.
root@192.168.1.47's password:
```

Once on the target, use these steps to create a trace:

```
root@crownbay:~# lttng create
Spawning a session daemon
Session auto-20121015-232120 created.
Traces will be written in /home/root/lttng-traces/auto-20121015-232120
```

Enable the events you want to trace (in this case all kernel events):

```
root@crownbay:~# lttng enable-event --kernel --all
All kernel events are enabled in channel channel0
```

Start the trace:

```
root@crownbay:~# lttng start
Tracing started for session auto-20121015-232120
```

And then stop the trace after awhile or after running a particular workload that you want to trace:

```
root@crownbay:~# lttng stop
Tracing stopped for session auto-20121015-232120
```

You can now view the trace in text form on the target:

```

root@crownbay:~# lttng view
[23:21:56.989270399] (+?.????????) sys_geteuid: { 1 }, { }
[23:21:56.989278081] (+0.000007682) exit_syscall: { 1 }, { ret = 0 }
[23:21:56.989286043] (+0.000007962) sys_pipe: { 1 }, { fildes = 0xB77B9E8C }
[23:21:56.989321802] (+0.000035759) exit_syscall: { 1 }, { ret = 0 }
[23:21:56.989329345] (+0.000007543) sys_mmap_pgoff: { 1 }, { addr = 0x0, len = 10485760, prot = 3, flags = 131362, fd = 4.
[23:21:56.989351694] (+0.000022349) exit_syscall: { 1 }, { ret = -1247805440 }
[23:21:56.989432989] (+0.000081295) sys_clone: { 1 }, { clone_flags = 0x411, newsp = 0xB5EFFFFE4, parent_tid = 0xFFFFFFFF,
[23:21:56.989477129] (+0.000044140) sched_stat_runtime: { 1 }, { comm = "lttng-consumerd", tid = 1193, runtime = 681660,
[23:21:56.989486697] (+0.000009568) sched_migrate_task: { 1 }, { comm = "lttng-consumerd", tid = 1193, prio = 20, orig_cp
[23:21:56.989508418] (+0.000021721) hrtimer_init: { 1 }, { hrtimer = 3970832076, clockid = 1, mode = 1 }
[23:21:56.989770462] (+0.000262044) hrtimer_cancel: { 1 }, { hrtimer = 3993865440 }
[23:21:56.989771580] (+0.000001118) hrtimer_cancel: { 0 }, { hrtimer = 3993812192 }
[23:21:56.989776957] (+0.000005377) hrtimer_expire_entry: { 1 }, { hrtimer = 3993865440, now = 79815980007057, function =
[23:21:56.989778145] (+0.000001188) hrtimer_expire_entry: { 0 }, { hrtimer = 3993812192, now = 79815980008174, function =
[23:21:56.989791695] (+0.000013550) softirq_raise: { 1 }, { vec = 1 }
[23:21:56.989795396] (+0.000003701) softirq_raise: { 0 }, { vec = 1 }
[23:21:56.989800635] (+0.000005239) softirq_raise: { 0 }, { vec = 9 }
[23:21:56.989807130] (+0.000006495) sched_stat_runtime: { 1 }, { comm = "lttng-consumerd", tid = 1193, runtime = 330710,
[23:21:56.989809993] (+0.000002863) sched_stat_runtime: { 0 }, { comm = "lttng-sessiond", tid = 1181, runtime = 1015313,
[23:21:56.989818514] (+0.000008521) hrtimer_expire_exit: { 0 }, { hrtimer = 3993812192 }
[23:21:56.989819631] (+0.000001117) hrtimer_expire_exit: { 1 }, { hrtimer = 3993865440 }
[23:21:56.989821866] (+0.000002235) hrtimer_start: { 0 }, { hrtimer = 3993812192, function = 3238465232, expires = 798159
[23:21:56.989822984] (+0.000001118) hrtimer_start: { 1 }, { hrtimer = 3993865440, function = 3238465232, expires = 798159
[23:21:56.989832762] (+0.000009778) softirq_entry: { 1 }, { vec = 1 }
[23:21:56.989833879] (+0.000001117) softirq_entry: { 0 }, { vec = 1 }
[23:21:56.989838069] (+0.000004190) timer_cancel: { 1 }, { timer = 3993871956 }
[23:21:56.989839187] (+0.000001118) timer_cancel: { 0 }, { timer = 3993818708 }
[23:21:56.989841492] (+0.000002305) timer_expire_entry: { 1 }, { timer = 3993871956, now = 79515980, function = 323827755
[23:21:56.989842819] (+0.000001327) timer_expire_entry: { 0 }, { timer = 3993818708, now = 79515980, function = 323827755
[23:21:56.989854831] (+0.000012012) sched_stat_runtime: { 1 }, { comm = "lttng-consumerd", tid = 1193, runtime = 49237, v
[23:21:56.989855949] (+0.000001118) sched_stat_runtime: { 0 }, { comm = "lttng-sessiond", tid = 1181, runtime = 45121, v
[23:21:56.989861257] (+0.000005308) sched_stat_sleep: { 1 }, { comm = "kworker/1:1", tid = 21, delay = 9451318 }
[23:21:56.989862374] (+0.000001117) sched_stat_sleep: { 0 }, { comm = "kworker/0:0", tid = 4, delay = 9958820 }
[23:21:56.989868241] (+0.000005867) sched_wakeup: { 0 }, { comm = "kworker/0:0", tid = 4, prio = 120, success = 1, target
[23:21:56.989869358] (+0.000001117) sched_wakeup: { 1 }, { comm = "kworker/1:1", tid = 21, prio = 120, success = 1, target
[23:21:56.989877460] (+0.000008102) timer_expire_exit: { 1 }, { timer = 3993871956 }
[23:21:56.989878577] (+0.000001117) timer_expire_exit: { 0 }, { timer = 3993818708 }
.
.
.

```

You can now safely destroy the trace session (note that this doesn't delete the trace - it's still there in ~/lttng-traces):

```

root@crownbay:~# lttng destroy
Session auto-20121015-232120 destroyed at /home/root

```

Note that the trace is saved in a directory of the same name as returned by 'lttng create', under the ~/lttng-traces directory (note that you can change this by supplying your own name to 'lttng create'):

```

root@crownbay:~# ls -al ~/lttng-traces
drwxrwx--- 3 root root 1024 Oct 15 23:21 .
drwxr-xr-x 5 root root 1024 Oct 15 23:57 ..
drwxrwx--- 3 root root 1024 Oct 15 23:21 auto-20121015-232120

```

### 19.5.2.2. Collecting and viewing a userspace trace on the target (inside a shell)

For LTTng userspace tracing, you need to have a properly instrumented userspace program. For this example, we'll use the 'hello' test program generated by the lttng-ust build.

The 'hello' test program isn't installed on the rootfs by the lttng-ust build, so we need to copy it over manually. First cd into the build directory that contains the hello executable:

```
$ cd build/tmp/work/core2_32-poky-linux/lttng-ust/2.0.5-r0/git/tests/hello/.libs
```

Copy that over to the target machine:

```
$ scp hello root@192.168.1.20:
```

You now have the instrumented lttng 'hello world' test program on the target, ready to test.

First, from the host, ssh to the target:

```

$ ssh -l root 192.168.1.47
The authenticity of host '192.168.1.47 (192.168.1.47)' can't be established.
RSA key fingerprint is 23:bd:c8:b1:a8:71:52:00:ee:00:4f:64:9e:10:b9:7e.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.1.47' (RSA) to the list of known hosts.
root@192.168.1.47's password:

```

Once on the target, use these steps to create a trace:

```

root@crownbay:~# lttng create
Session auto-20190303-021943 created.
Traces will be written in /home/root/lttng-traces/auto-20190303-021943

```

Enable the events you want to trace (in this case all userspace events):

```

root@crownbay:~# lttng enable-event --userspace --all
All UST events are enabled in channel channel0

```

Start the trace:

```
root@crownbay:~# lttng start
Tracing started for session auto-20190303-021943
```

Run the instrumented hello world program:

```
root@crownbay:~# ./hello
Hello, World!
Tracing... done.
```

And then stop the trace after awhile or after running a particular workload that you want to trace:

```
root@crownbay:~# lttng stop
Tracing stopped for session auto-20190303-021943
```

You can now view the trace in text form on the target:

```
root@crownbay:~# lttng view
[02:31:14.906146544] (+?.????????) hello:1424 ust_tests_hello:tpctest: { cpu_id = 1 }, { intfield = 0, intfield2 = 0x0, l
[02:31:14.906170360] (+0.000023816) hello:1424 ust_tests_hello:tpctest: { cpu_id = 1 }, { intfield = 1, intfield2 = 0x1, l
[02:31:14.906183140] (+0.000012780) hello:1424 ust_tests_hello:tpctest: { cpu_id = 1 }, { intfield = 2, intfield2 = 0x2, l
[02:31:14.906194385] (+0.000011245) hello:1424 ust_tests_hello:tpctest: { cpu_id = 1 }, { intfield = 3, intfield2 = 0x3, l
.
.
.
```

You can now safely destroy the trace session (note that this doesn't delete the trace - it's still there in ~/lttng-traces):

```
root@crownbay:~# lttng destroy
Session auto-20190303-021943 destroyed at /home/root
```

### 19.5.3. Documentation¶

You can find the primary LTTng Documentation on the [LTTng Documentation](#) site. The documentation on this site is appropriate for intermediate to advanced software developers who are working in a Linux environment and are interested in efficient software tracing.

For information on LTTng in general, visit the [LTTng Project](#) site. You can find a "Getting Started" link on this site that takes you to an LTTng Quick Start.

## 19.6. blktrace¶

blktrace is a tool for tracing and reporting low-level disk I/O. blktrace provides the tracing half of the equation; its output can be piped into the blkparse program, which renders the data in a human-readable form and does some basic analysis:

### 19.6.1. Setup¶

For this section, we'll assume you've already performed the basic setup outlined in the "[General Setup](#)" section.

blktrace is an application that runs on the target system. You can run the entire blktrace and blkparse pipeline on the target, or you can run blktrace in 'listen' mode on the target and have blktrace and blkparse collect and analyze the data on the host (see the "[Using blktrace Remotely](#)," section below). For the rest of this section we assume you've ssh'ed to the host and will be running blktrace on the target.

### 19.6.2. Basic Usage¶

To record a trace, simply run the 'blktrace' command, giving it the name of the block device you want to trace activity on:

```
root@crownbay:~# blktrace /dev/sdc
```

In another shell, execute a workload you want to trace.

```
root@crownbay:/media/sdc# rm linux-2.6.19.2.tar.bz2; wget http://downloads.yoctoproject.org/mirror/sources/linux-2.6.19.2
Connecting to downloads.yoctoproject.org (140.211.169.59:80)
linux-2.6.19.2.tar.b 100% |*****| 41727k 0:00:00 ETA
```

Press Ctrl-C in the blktrace shell to stop the trace. It will display how many events were logged, along with the per-cpu file sizes (blktrace records traces in per-cpu kernel buffers and simply dumps them to userspace for blkparse to merge and sort later).

```
^C=== sdc ===
CPU 0: 7082 events, 332 KiB data
CPU 1: 1578 events, 74 KiB data
Total: 8660 events (dropped 0), 406 KiB data
```

If you examine the files saved to disk, you see multiple files, one per CPU and with the device name as the first part of the filename:

```
root@crownbay:~# ls -al
drwxr-xr-x 6 root root 1024 Oct 27 22:39 .
drwxr-sr-x 4 root root 1024 Oct 26 18:24 ..
-rw-r--r-- 1 root root 339938 Oct 27 22:40 sdc.blktrace.0
-rw-r--r-- 1 root root 75753 Oct 27 22:40 sdc.blktrace.1
```

To view the trace events, simply invoke 'blkparse' in the directory containing the trace files, giving it the device name that forms the first part of the filenames:

```
root@crownbay:~# blkparse sdc
```

```

8,32 1 1 0.000000000 1225 Q WS 3417048 + 8 [jbd2/sdc-8]
8,32 1 2 0.000025213 1225 G WS 3417048 + 8 [jbd2/sdc-8]
8,32 1 3 0.000033384 1225 P N [jbd2/sdc-8]
8,32 1 4 0.000043301 1225 I WS 3417048 + 8 [jbd2/sdc-8]
8,32 1 0 0.000057270 0 m N cfq1225 insert_request
8,32 1 0 0.000064813 0 m N cfq1225 add_to_rr
8,32 1 5 0.000076336 1225 U N [jbd2/sdc-8] 1
8,32 1 0 0.000088559 0 m N cfq workload slice:150
8,32 1 0 0.000097359 0 m N cfq1225 set_active wl_prio:0 wl_type:1
8,32 1 0 0.000104063 0 m N cfq1225 Not idling. st->count:1
8,32 1 0 0.000112584 0 m N cfq1225 fifo= (null)
8,32 1 0 0.000118730 0 m N cfq1225 dispatch_insert
8,32 1 0 0.000127390 0 m N cfq1225 dispatched a request
8,32 1 0 0.000133536 0 m N cfq1225 activate rq, drv=1
8,32 1 6 0.000136889 1225 D WS 3417048 + 8 [jbd2/sdc-8]
8,32 1 7 0.000360381 1225 Q WS 3417056 + 8 [jbd2/sdc-8]
8,32 1 8 0.000377422 1225 G WS 3417056 + 8 [jbd2/sdc-8]
8,32 1 9 0.000388876 1225 P N [jbd2/sdc-8]
8,32 1 10 0.000397886 1225 Q WS 3417064 + 8 [jbd2/sdc-8]
8,32 1 11 0.000404800 1225 M WS 3417064 + 8 [jbd2/sdc-8]
8,32 1 12 0.000412343 1225 Q WS 3417072 + 8 [jbd2/sdc-8]
8,32 1 13 0.000416533 1225 M WS 3417072 + 8 [jbd2/sdc-8]
8,32 1 14 0.000422121 1225 Q WS 3417080 + 8 [jbd2/sdc-8]
8,32 1 15 0.000425194 1225 M WS 3417080 + 8 [jbd2/sdc-8]
8,32 1 16 0.000431968 1225 Q WS 3417088 + 8 [jbd2/sdc-8]
8,32 1 17 0.000435251 1225 M WS 3417088 + 8 [jbd2/sdc-8]
8,32 1 18 0.000440279 1225 Q WS 3417096 + 8 [jbd2/sdc-8]
8,32 1 19 0.000443911 1225 M WS 3417096 + 8 [jbd2/sdc-8]
8,32 1 20 0.000450336 1225 Q WS 3417104 + 8 [jbd2/sdc-8]
8,32 1 21 0.000454038 1225 M WS 3417104 + 8 [jbd2/sdc-8]
8,32 1 22 0.000462070 1225 Q WS 3417112 + 8 [jbd2/sdc-8]
8,32 1 23 0.000465422 1225 M WS 3417112 + 8 [jbd2/sdc-8]
8,32 1 24 0.000474222 1225 I WS 3417056 + 64 [jbd2/sdc-8]
8,32 1 0 0.000483022 0 m N cfq1225 insert_request
8,32 1 25 0.000489727 1225 U N [jbd2/sdc-8] 1
8,32 1 0 0.000498457 0 m N cfq1225 Not idling. st->count:1
8,32 1 0 0.000503765 0 m N cfq1225 dispatch_insert
8,32 1 0 0.000512914 0 m N cfq1225 dispatched a request
8,32 1 0 0.000518851 0 m N cfq1225 activate rq, drv=2
.
.
.
8,32 0 0 58.515006138 0 m N cfq3551 complete rqnoidle 1
8,32 0 2024 58.516603269 3 C WS 3156992 + 16 [0]
8,32 0 0 58.516626736 0 m N cfq3551 complete rqnoidle 1
8,32 0 0 58.516634558 0 m N cfq3551 arm_idle: 8 group_idle: 0
8,32 0 0 58.516636933 0 m N cfq schedule dispatch
8,32 1 0 58.516971613 0 m N cfq3551 slice expired t=0
8,32 1 0 58.516982089 0 m N cfq3551 sl_used=13 disp=6 charge=13 iops=0 sect=80
8,32 1 0 58.516985511 0 m N cfq3551 del_from_rr
8,32 1 0 58.516990819 0 m N cfq3551 put_queue

CPU0 (sdc):
Reads Queued: 0, 0KiB Writes Queued: 331, 26,284KiB
Read Dispatches: 0, 0KiB Write Dispatches: 485, 40,484KiB
Reads Requeued: 0 Writes Requeued: 0
Reads Completed: 0, 0KiB Writes Completed: 511, 41,000KiB
Read Merges: 0, 0KiB Write Merges: 13, 160KiB
Read depth: 0 Write depth: 2
IO unplugs: 23 Timer unplugs: 0

CPU1 (sdc):
Reads Queued: 0, 0KiB Writes Queued: 249, 15,800KiB
Read Dispatches: 0, 0KiB Write Dispatches: 42, 1,600KiB
Reads Requeued: 0 Writes Requeued: 0
Reads Completed: 0, 0KiB Writes Completed: 16, 1,084KiB
Read Merges: 0, 0KiB Write Merges: 40, 276KiB
Read depth: 0 Write depth: 2
IO unplugs: 30 Timer unplugs: 1

Total (sdc):
Reads Queued: 0, 0KiB Writes Queued: 580, 42,084KiB
Read Dispatches: 0, 0KiB Write Dispatches: 527, 42,084KiB
Reads Requeued: 0 Writes Requeued: 0
Reads Completed: 0, 0KiB Writes Completed: 527, 42,084KiB
Read Merges: 0, 0KiB Write Merges: 53, 436KiB
IO unplugs: 53 Timer unplugs: 1

Throughput (R/W): 0KiB/s / 719KiB/s
Events (sdc): 6,592 entries
Skips: 0 forward (0 - 0.0%)
Input file sdc.blktrace.0 added
Input file sdc.blktrace.1 added

```

The report shows each event that was found in the blktrace data, along with a summary of the overall block I/O traffic during the run. You can look at the [blkparse](#) manpage to learn the meaning of each field displayed in the trace listing.

### 19.6.2.1. Live Mode¶

blktrace and blkparse are designed from the ground up to be able to operate together in a 'pipe mode' where the stdout of blktrace can be fed directly into the stdin of blkparse:

```
root@crownbay:~# blktrace /dev/sdc -o - | blkparse -i -
```

This enables long-lived tracing sessions to run without writing anything to disk, and allows the user to look for certain conditions in the trace data in 'real-time' by viewing the trace output as it scrolls by on the screen or by passing it along to yet another program in the pipeline such as grep which can be used to identify and capture conditions of interest.

There's actually another blktrace command that implements the above pipeline as a single command, so the user doesn't have to bother typing in the above command sequence:

```
root@crownbay:~# btrace /dev/sdc
```

### 19.6.2.2. Using blktrace Remotely

Because blktrace traces block I/O and at the same time normally writes its trace data to a block device, and in general because it's not really a great idea to make the device being traced the same as the device the tracer writes to, blktrace provides a way to trace without perturbing the traced device at all by providing native support for sending all trace data over the network.

To have blktrace operate in this mode, start blktrace on the target system being traced with the `-l` option, along with the device to trace:

```
root@crownbay:~# blktrace -l /dev/sdc
server: waiting for connections...
```

On the host system, use the `-h` option to connect to the target system, also passing it the device to trace:

```
$ blktrace -d /dev/sdc -h 192.168.1.43
blktrace: connecting to 192.168.1.43
blktrace: connected!
```

On the target system, you should see this:

```
server: connection from 192.168.1.43
```

In another shell, execute a workload you want to trace.

```
root@crownbay:/media/sdc# rm linux-2.6.19.2.tar.bz2; wget http://downloads.yoctoproject.org/mirror/sources/linux-2.6.19.2
Connecting to downloads.yoctoproject.org (140.211.169.59:80)
linux-2.6.19.2.tar.b 100% |*****| 41727k 0:00:00 ETA
```

When it's done, do a Ctrl-C on the host system to stop the trace:

```
^C=== sdc ===
CPU 0: 7691 events, 361 KiB data
CPU 1: 4109 events, 193 KiB data
Total: 11800 events (dropped 0), 554 KiB data
```

On the target system, you should also see a trace summary for the trace just ended:

```
server: end of run for 192.168.1.43:sdc
=== sdc ===
CPU 0: 7691 events, 361 KiB data
CPU 1: 4109 events, 193 KiB data
Total: 11800 events (dropped 0), 554 KiB data
```

The blktrace instance on the host will save the target output inside a hostname-timestamp directory:

```
$ ls -al
drwxr-xr-x 10 root root 1024 Oct 28 02:40 .
drwxr-sr-x 4 root root 1024 Oct 26 18:24 ..
drwxr-xr-x 2 root root 1024 Oct 28 02:40 192.168.1.43-2012-10-28-02:40:56
```

cd into that directory to see the output files:

```
$ ls -l
-rw-r--r-- 1 root root 369193 Oct 28 02:44 sdc.blktrace.0
-rw-r--r-- 1 root root 197278 Oct 28 02:44 sdc.blktrace.1
```

And run blkparse on the host system using the device name:

```
$ blkparse sdc

8,32 1 1 0.000000000 1263 Q RM 6016 + 8 [ls]
8,32 1 0 0.000036038 0 m N cfq1263 allocated
8,32 1 2 0.000039390 1263 G RM 6016 + 8 [ls]
8,32 1 3 0.000049168 1263 I RM 6016 + 8 [ls]
8,32 1 0 0.000056152 0 m N cfq1263 insert_request
8,32 1 0 0.000061600 0 m N cfq1263 add_to_rr
8,32 1 0 0.000075498 0 m N cfq workload slice:300
.
.
.
8,32 0 0 177.266385696 0 m N cfq1267 arm_idle: 8 group_idle: 0
8,32 0 0 177.266388140 0 m N cfq schedule dispatch
8,32 1 0 177.266679239 0 m N cfq1267 slice expired t=0
8,32 1 0 177.266689297 0 m N cfq1267 sl_used=9 disp=6 charge=9 iops=0 sect=56
8,32 1 0 177.266692649 0 m N cfq1267 del_from_rr
8,32 1 0 177.266696560 0 m N cfq1267 put_queue

CPU0 (sdc):
Reads Queued: 0, 0KiB Writes Queued: 270, 21,708KiB
Read Dispatches: 59, 2,628KiB Write Dispatches: 495, 39,964KiB
Reads Requeued: 0 Writes Requeued: 0
Reads Completed: 90, 2,752KiB Writes Completed: 543, 41,596KiB
Read Merges: 0, 0KiB Write Merges: 9, 344KiB
Read depth: 2 Write depth: 2
IO unplugs: 20 Timer unplugs: 1

CPU1 (sdc):
Reads Queued: 688, 2,752KiB Writes Queued: 381, 20,652KiB
Read Dispatches: 31, 124KiB Write Dispatches: 59, 2,396KiB
```

```

Reads Requeued: 0 Writes Requeued: 0
Reads Completed: 0, 0KiB Writes Completed: 11, 764KiB
Read Merges: 598, 2,392KiB Write Merges: 88, 448KiB
Read depth: 2 Write depth: 2
IO unplugs: 52 Timer unplugs: 0

Total (sdc):
Reads Queued: 688, 2,752KiB Writes Queued: 651, 42,360KiB
Read Dispatches: 90, 2,752KiB Write Dispatches: 554, 42,360KiB
Reads Requeued: 0 Writes Requeued: 0
Reads Completed: 90, 2,752KiB Writes Completed: 554, 42,360KiB
Read Merges: 598, 2,392KiB Write Merges: 97, 792KiB
IO unplugs: 72 Timer unplugs: 1

Throughput (R/W): 15KiB/s / 238KiB/s
Events (sdc): 9,301 entries
Skips: 0 forward (0 - 0.0%)

```

You should see the trace events and summary just as you would have if you'd run the same command on the target.

### 19.6.2.3. Tracing Block I/O via 'ftrace'

It's also possible to trace block I/O using only [trace\\_events subsystem](#), which can be useful for casual tracing if you don't want to bother dealing with the userspace tools.

To enable tracing for a given device, use `/sys/block/xxx/trace/enable`, where `xxx` is the device name. This for example enables tracing for `/dev/sdc`:

```
root@crownbay:/sys/kernel/debug/tracing# echo 1 > /sys/block/sdc/trace/enable
```

Once you've selected the device(s) you want to trace, selecting the 'blk' tracer will turn the blk tracer on:

```
root@crownbay:/sys/kernel/debug/tracing# cat available_tracers
blk function_graph function nop

root@crownbay:/sys/kernel/debug/tracing# echo blk > current_tracer
```

Execute the workload you're interested in:

```
root@crownbay:/sys/kernel/debug/tracing# cat /media/sdc/testfile.txt
```

And look at the output (note here that we're using 'trace\_pipe' instead of trace to capture this trace - this allows us to wait around on the pipe for data to appear):

```

root@crownbay:/sys/kernel/debug/tracing# cat trace_pipe
cat-3587 [001] d..1 3023.276361: 8,32 Q R 1699848 + 8 [cat]
cat-3587 [001] d..1 3023.276410: 8,32 m N cfq3587 allocated
cat-3587 [001] d..1 3023.276415: 8,32 G R 1699848 + 8 [cat]
cat-3587 [001] d..1 3023.276424: 8,32 P N [cat]
cat-3587 [001] d..2 3023.276432: 8,32 I R 1699848 + 8 [cat]
cat-3587 [001] d..1 3023.276439: 8,32 m N cfq3587 insert_request
cat-3587 [001] d..1 3023.276445: 8,32 m N cfq3587 add_to_rr
cat-3587 [001] d..2 3023.276454: 8,32 U N [cat] 1
cat-3587 [001] d..1 3023.276464: 8,32 m N cfq workload slice:150
cat-3587 [001] d..1 3023.276471: 8,32 m N cfq3587 set_active wl_prio:0 wl_type:2
cat-3587 [001] d..1 3023.276478: 8,32 m N cfq3587 fifo= (null)
cat-3587 [001] d..1 3023.276483: 8,32 m N cfq3587 dispatch_insert
cat-3587 [001] d..1 3023.276490: 8,32 m N cfq3587 dispatched a request
cat-3587 [001] d..1 3023.276497: 8,32 m N cfq3587 activate rq, drv=1
cat-3587 [001] d..2 3023.276500: 8,32 D R 1699848 + 8 [cat]

```

And this turns off tracing for the specified device:

```
root@crownbay:/sys/kernel/debug/tracing# echo 0 > /sys/block/sdc/trace/enable
```

### 19.6.3. Documentation

Online versions of the man pages for the commands discussed in this section can be found here:

- <http://linux.die.net/man/8/blktrace>
- <http://linux.die.net/man/1/blkparse>
- <http://linux.die.net/man/8/btrace>

The above manpages, along with manpages for the other blktrace utilities (btt, blkionmon, etc) can be found in the `/doc` directory of the blktrace tools git repo:

```
$ git clone git://git.kernel.dk/blktrace.git
```

## Chapter 20. Real-World Examples

### 20.1. Slow Write Speed on Live Images

This chapter contains real-world examples.



## 20.1. Slow Write Speed on Live Images¶

In one of our previous releases (denzil), users noticed that booting off of a live image and writing to disk was noticeably slower. This included the boot itself, especially the first one, since first boots tend to do a significant amount of writing due to certain post-install scripts.

The problem (and solution) was discovered by using the Yocto tracing tools, in this case 'perf stat', 'perf script', 'perf record' and 'perf report'.

See all the unvarnished details of how this bug was diagnosed and solved here: [Yocto Bug #3049](#)



# Yocto Project Reference Manual

## Chapter 21. System Requirements¶

### [21.1. Supported Linux Distributions](#)

#### [21.2. Required Packages for the Build Host](#)

##### [21.2.1. Ubuntu and Debian](#)

##### [21.2.2. Fedora Packages](#)

##### [21.2.3. openSUSE Packages](#)

##### [21.2.4. CentOS Packages](#)

### [21.3. Required Git, tar, and Python Versions](#)

#### [21.3.1. Downloading a Pre-Built `buildtools` Tarball](#)

#### [21.3.2. Building Your Own `buildtools` Tarball](#)

Welcome to the Yocto Project Reference Manual! This manual provides reference information for the current release of the Yocto Project. The manual is best used after you have an understanding of the basics of the Yocto Project. The manual is neither meant to be read as a starting point to the Yocto Project nor read from start to finish. Use this manual to find variable definitions, class descriptions, and so forth as needed during the course of using the Yocto Project.

For introductory information on the Yocto Project, see the [Yocto Project Website](#) and the "[Yocto Project Development Environment](#)" chapter in the Yocto Project Overview and Concepts Manual.

If you want to use the Yocto Project to quickly build an image without having to understand concepts, work through the Yocto Project Quick Build document. You can find "how-to" information in the Yocto Project Development Tasks Manual. You can find Yocto Project overview and conceptual information in the Yocto project Overview and Concepts Manual.

### Tip

For more information about the Yocto Project Documentation set, see the "[Links and Related Documentation](#)" section.

## 21.1. Supported Linux Distributions¶

Currently, the Yocto Project is supported on the following distributions:

### Notes

- Yocto Project releases are tested against the stable Linux distributions in the following list. The Yocto Project should work on other distributions but validation is not performed against them.
- In particular, the Yocto Project does not support and currently has no plans to support rolling-releases or development distributions due to their constantly changing nature. We welcome patches and bug reports, but keep in mind that our priority is on the supported platforms listed below.
- The Yocto Project is not compatible with the [Windows Subsystem for Linux](#) (WSL). You cannot use a [build host](#) that is running WSL.
- If you encounter problems, please go to [Yocto Project Bugzilla](#) and submit a bug. We are interested in hearing about your experience. For information on how to submit a bug, see the Yocto Project [Bugzilla wiki page](#) and the "[Submitting a Defect Against the Yocto Project](#)" section in the Yocto Project Development Tasks Manual.

- Ubuntu 16.04 (LTS)
- Ubuntu 18.04 (LTS)
- Ubuntu 19.04
- Fedora 28



- Fedora 29
- Fedora 30
- CentOS 7.x
- Debian GNU/Linux 8.x (Jessie)
- Debian GNU/Linux 9.x (Stretch)
- Debian GNU/Linux 10.x (Buster)
- OpenSUSE Leap 15.1

### Note

While the Yocto Project Team attempts to ensure all Yocto Project releases are one hundred percent compatible with each officially supported Linux distribution, instances might exist where you encounter a problem while using the Yocto Project on a specific distribution.

## 21.2. Required Packages for the Build Host¶

The list of packages you need on the host development system can be large when covering all build scenarios using the Yocto Project. This section provides required packages according to Linux distribution and function.

### 21.2.1. Ubuntu and Debian¶

The following list shows the required packages by function given a supported Ubuntu or Debian Linux distribution:

### Note

If your build system has the `oss4-dev` package installed, you might experience QEMU build failures due to the package installing its own custom `/usr/include/linux/soundcard.h` on the Debian system. If you run into this situation, either of the following solutions exist:

```
$ sudo apt-get build-dep qemu
$ sudo apt-get remove oss4-dev
```

- **Essentials:** Packages needed to build an image on a headless system:

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib \
build-essential chrpath socat cpio python python3 python3-pip python3-pexpect \
xz-utils debianutils iputils-ping python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev \
pylint3 xterm
```

- **Documentation:** Packages needed if you are going to build out the Yocto Project documentation manuals:

```
$ sudo apt-get install make xsltproc docbook-utils fop dblatex xmlto
```

### 21.2.2. Fedora Packages¶

The following list shows the required packages by function given a supported Fedora Linux distribution:

- **Essentials:** Packages needed to build an image for a headless system:

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

- **Documentation:** Packages needed if you are going to build out the Yocto Project documentation manuals:

```
$ sudo dnf install docbook-style-dsssl docbook-style-xsl \
docbook-dtds docbook-utils fop libxslt dblatex xmlto
```

### 21.2.3. openSUSE Packages¶

The following list shows the required packages by function given a supported openSUSE Linux distribution:

- **Essentials:** Packages needed to build an image for a headless system:

```
$ sudo zypper install python gcc gcc-c++ git chrpath make wget python-xml \
diffstat makeinfo python-curses patch socat python3 python3-pip \
python3-pexpect xz which python3-Jinja2 Mesa-libEGL1
$ sudo pip3 install GitPython libSDL-devel xterm
```

- **Documentation:** Packages needed if you are going to build out the Yocto Project documentation manuals:

```
$ sudo zypper install dblatex xmlto
```

### 21.2.4. CentOS Packages¶

The following list shows the required packages by function given a supported CentOS Linux distribution:

- **Essentials:** Packages needed to build an image for a headless system:

```
$ sudo yum install -y epel-release
$ sudo yum makecache
$ 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 python34-pip xz \
which SDL-devel xterm
$ sudo pip3 install GitPython Jinja2
```

#### Notes

- Extra Packages for Enterprise Linux (i.e. `epel-release`) is a collection of packages from Fedora built on RHEL/CentOS for easy installation of packages not included in enterprise Linux by default. You need to install these packages separately.
- The `makecache` command consumes additional Metadata from `epel-release`.

- **Documentation:** Packages needed if you are going to build out the Yocto Project documentation manuals:

```
$ sudo yum install docbook-style-dsssl docbook-style-xsl \
docbook-dtds docbook-utils fop libxslt dblatex xmlto
```

### 21.3. Required Git, tar, and Python Versions¶

In order to use the build system, your host development system must meet the following version requirements for Git, tar, and Python:

- Git 1.8.3.1 or greater
- tar 1.27 or greater
- Python 3.4.0 or greater

If your host development system does not meet all these requirements, you can resolve this by installing a `buildtools` tarball that contains these tools. You can get the tarball one of two ways: download a pre-built tarball or use BitBake to build the tarball.

#### 21.3.1. Downloading a Pre-Built `buildtools` Tarball¶

Downloading and running a pre-built `buildtools` installer is the easiest of the two methods by which you can get these tools:

1. Locate and download the `*.sh` at <http://downloads.yoctoproject.org/releases/yocto/yocto-3.0.1/buildtools/>.
2. Execute the installation script. Here is an example:

```
$ sh ~/Downloads/x86_64-buildtools-nativesdk-standalone-3.0.1.sh
```

During execution, a prompt appears that allows you to choose the installation directory. For example, you could choose the following:

```
/home/your-username/buildtools
```

3. Source the tools environment setup script by using a command like the following:

```
$ source /home/your_username/buildtools/environment-setup-i586-poky-linux
```

Of course, you need to supply your installation directory and be sure to use the right file (i.e. `i586` or `x86-64`).

After you have sourced the setup script, the tools are added to `PATH` and any other environment variables required to run the tools are initialized. The results are working versions of Git, tar, Python and `chrpath`.

#### 21.3.2. Building Your Own `buildtools` Tarball¶

Building and running your own `buildtools` installer applies only when you have a build host that can already run BitBake. In this case, you use that machine to build the `.sh` file and then take steps to transfer and run it on a machine that does not meet the minimal Git, tar, and Python requirements.

Here are the steps to take to build and run your own `buildtools` installer:

1. On the machine that is able to run BitBake, be sure you have set up your build environment with the setup script (`oe-init-build-env`).
2. Run the BitBake command to build the tarball:

```
$ bitbake buildtools-tarball
```

#### Note

The `SDKMACHINE` variable in your `local.conf` file determines whether you build tools for a 32-bit or 64-bit system.

Once the build completes, you can find the `.sh` file that installs the tools in the `tmp/deploy/sdk` subdirectory of the [Build Directory](#). The installer file has the string "buildtools" in the name.

- Transfer the `.sh` file from the build host to the machine that does not meet the Git, tar, or Python requirements.
- On the machine that does not meet the requirements, run the `.sh` file to install the tools. Here is an example:

```
$ sh ~/Downloads/x86_64-buildtools-nativesdk-standalone-3.0.1.sh
```

During execution, a prompt appears that allows you to choose the installation directory. For example, you could choose the following:

```
/home/your_username/buildtools
```

- Source the tools environment setup script by using a command like the following:

```
$ source /home/your_username/buildtools/environment-setup-i586-poky-linux
```

Of course, you need to supply your installation directory and be sure to use the right file (i.e. i585 or x86-64).

After you have sourced the setup script, the tools are added to `PATH` and any other environment variables required to run the tools are initialized. The results are working versions versions of Git, tar, Python and `chrpath`.

## Chapter 22. Yocto Project Terms¶

Following is a list of terms and definitions users new to the Yocto Project development environment might find helpful. While some of these terms are universal, the list includes them just in case:

- **Append Files:** Files that append build information to a recipe file. Append files are known as BitBake append files and `.bbappend` files. The OpenEmbedded build system expects every append file to have a corresponding recipe (`.bb`) file. Furthermore, the append file and corresponding recipe file must use the same root filename. The filenames can differ only in the file type suffix used (e.g. `formfactor_0.0.bb` and `formfactor_0.0.bbappend`).

Information in append files extends or overrides the information in the similarly-named recipe file. For an example of an append file in use, see the ["Using .bbappend Files in Your Layer"](#) section in the Yocto Project Development Tasks Manual.

When you name an append file, you can use the `"%"` wildcard character to allow for matching recipe names. For example, suppose you have an append file named as follows:

```
busybox_1.21.%.bbappend
```

That append file would match any `busybox_1.21.x.bb` version of the recipe. So, the append file would match the following recipe names:

```
busybox_1.21.1.bb
busybox_1.21.2.bb
busybox_1.21.3.bb
```

### Important

The use of the `"%"` character is limited in that it only works directly in front of the `.bbappend` portion of the append file's name. You cannot use the wildcard character in any other location of the name.

- **BitBake:** The task executor and scheduler used by the OpenEmbedded build system to build images. For more information on BitBake, see the BitBake User Manual.
- **Board Support Package (BSP):** A group of drivers, definitions, and other components that provide support for a specific hardware configuration. For more information on BSPs, see the Yocto Project Board Support Package (BSP) Developer's Guide.
- **Build Directory:** This term refers to the area used by the OpenEmbedded build system for builds. The area is created when you `source` the setup environment script that is found in the Source Directory (i.e. `oe-init-build-env`). The `TOPDIR` variable points to the Build Directory.

You have a lot of flexibility when creating the Build Directory. Following are some examples that show how to create the directory. The examples assume your [Source Directory](#) is named `poky`:

- Create the Build Directory inside your Source Directory and let the name of the Build Directory default to `build`:

```
$ cd $HOME/poky
$ source oe-init-build-env
```

- Create the Build Directory inside your home directory and specifically name it `test-builds`:

```
$ cd $HOME
$ source poky/oe-init-build-env test-builds
```

- Provide a directory path and specifically name the Build Directory. Any intermediate folders in the pathname must exist. This next example creates a Build Directory named `YP-22.0.1` in your home directory within the existing directory `mybuilds`:

```
$cd $HOME
$ source $HOME/poky/oe-init-build-env $HOME/mybuilds/YP-22.0.1
```

## Note

By default, the Build Directory contains `TMPDIR`, which is a temporary directory the build system uses for its work. `TMPDIR` cannot be under NFS. Thus, by default, the Build Directory cannot be under NFS. However, if you need the Build Directory to be under NFS, you can set this up by setting `TMPDIR` in your `local.conf` file to use a local drive. Doing so effectively separates `TMPDIR` from `TOPDIR`, which is the Build Directory.

- **Build Host:** The system used to build images in a Yocto Project Development environment. The build system is sometimes referred to as the development host.
- **Classes:** Files that provide for logic encapsulation and inheritance so that commonly used patterns can be defined once and then easily used in multiple recipes. For reference information on the Yocto Project classes, see the "[Classes](#)" chapter. Class files end with the `.bbclass` filename extension.
- **Configuration File:** Files that hold global definitions of variables, user-defined variables, and hardware configuration information. These files tell the OpenEmbedded build system what to build and what to put into the image to support a particular platform.

Configuration files end with a `.conf` filename extension. The `conf/local.conf` configuration file in the [Build Directory](#) contains user-defined variables that affect every build. The `meta-poky/conf/distro/poky.conf` configuration file defines Yocto "distro" configuration variables used only when building with this policy. Machine configuration files, which are located throughout the [Source Directory](#), define variables for specific hardware and are only used when building for that target (e.g. the `machine/beaglebone.conf` configuration file defines variables for the Texas Instruments ARM Cortex-A8 development board).

- **Container Layer:** Layers that hold other layers. An example of a container layer is OpenEmbedded's [meta-openembedded](#) layer. The `meta-openembedded` layer contains many `meta-*` layers.
- **Cross-Development Toolchain:** In general, a cross-development toolchain is a collection of software development tools and utilities that run on one architecture and allow you to develop software for a different, or targeted, architecture. These toolchains contain cross-compilers, linkers, and debuggers that are specific to the target architecture.

The Yocto Project supports two different cross-development toolchains:

- A toolchain only used by and within BitBake when building an image for a target architecture.
- A relocatable toolchain used outside of BitBake by developers when developing applications that will run on a targeted device.

Creation of these toolchains is simple and automated. For information on toolchain concepts as they apply to the Yocto Project, see the "[Cross-Development Toolchain Generation](#)" section in the Yocto Project Overview and Concepts Manual. You can also find more information on using the relocatable toolchain in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

- **Extensible Software Development Kit (eSDK):** A custom SDK for application developers. This eSDK allows developers to incorporate their library and programming changes back into the image to make their code available to other application developers.

For information on the eSDK, see the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

- **Image:** An image is an artifact of the BitBake build process given a collection of recipes and related Metadata. Images are the binary output that run on specific hardware or QEMU and are used for specific use-cases. For a list of the supported image types that the Yocto Project provides, see the "[Images](#)" chapter.
- **Layer:** A collection of related recipes. Layers allow you to consolidate related metadata to customize your build. Layers also isolate information used when building for multiple architectures. Layers are hierarchical in their ability to override previous specifications. You can include any number of available layers from the Yocto Project and customize the build by adding your layers after them. You can search the Layer Index for layers used within Yocto Project.

For introductory information on layers, see the "[The Yocto Project Layer Model](#)" section in the Yocto Project Overview and Concepts Manual. For more detailed information on layers, see the "[Understanding and Creating Layers](#)" section in the Yocto Project Development Tasks Manual. For a discussion specifically on BSP Layers, see the "[BSP Layers](#)" section in the Yocto Project Board Support Packages (BSP) Developer's Guide.

- **Metadata:** A key element of the Yocto Project is the Metadata that is used to construct a Linux distribution and is contained in the files that the [OpenEmbedded build system](#) parses when building an image. In general, Metadata includes recipes, configuration files, and other information that refers to the build instructions themselves, as well as the data used to control what things get built and the effects of the build. Metadata also includes commands and data used to indicate what versions of software are used, from where they are obtained, and changes or additions to the software itself (patches or auxiliary files) that are used to fix bugs or customize the software for use in a particular situation. OpenEmbedded-Core is an important set of validated metadata.

In the context of the kernel ("kernel Metadata"), the term refers to the kernel config fragments and features contained in the [yocto-kernel-cache](#) Git repository.

- **OpenEmbedded-Core (OE-Core):** OE-Core is metadata comprised of foundational recipes, classes, and associated files that are meant to be common among many different OpenEmbedded-derived systems, including the Yocto Project. OE-Core is a curated subset of an original repository developed by the OpenEmbedded community that has been pared down into a smaller, core set of continuously validated recipes. The result is a tightly controlled and an quality-assured core set of recipes.

You can see the Metadata in the `meta` directory of the Yocto Project [Source Repositories](#).

- **OpenEmbedded Build System:** The build system specific to the Yocto Project. The OpenEmbedded build system is based on another project known as "Poky", which uses [BitBake](#) as the task executor. Throughout the Yocto Project documentation set, the OpenEmbedded build system is sometimes referred to simply as "the build system". If other build systems, such as a host or target build system are referenced, the documentation clearly states the difference.

## Note

For some historical information about Poky, see the [Poky](#) term.

- **Package:** In the context of the Yocto Project, this term refers to a recipe's packaged output produced by BitBake (i.e. a "baked recipe"). A package is generally the compiled binaries produced from the recipe's sources. You "bake" something by running it through BitBake.

It is worth noting that the term "package" can, in general, have subtle meanings. For example, the packages referred to in the ["Required Packages for the Build Host"](#) section are compiled binaries that, when installed, add functionality to your Linux distribution.

Another point worth noting is that historically within the Yocto Project, recipes were referred to as packages - thus, the existence of several BitBake variables that are seemingly mis-named, (e.g. [PR](#), [PV](#), and [PE](#)).

- **Package Groups:** Arbitrary groups of software Recipes. You use package groups to hold recipes that, when built, usually accomplish a single task. For example, a package group could contain the recipes for a company's proprietary or value-add software. Or, the package group could contain the recipes that enable graphics. A package group is really just another recipe. Because package group files are recipes, they end with the `.bb` filename extension.

- **Poky:** Poky, which is pronounced **Pock**-ee, is a reference embedded distribution and a reference test configuration. Poky provides the following:

- A base-level functional distro used to illustrate how to customize a distribution.
- A means by which to test the Yocto Project components (i.e. Poky is used to validate the Yocto Project).
- A vehicle through which you can download the Yocto Project.

Poky is not a product level distro. Rather, it is a good starting point for customization.

## Note

Poky began an open-source project initially developed by OpenedHand. OpenedHand developed Poky from the existing OpenEmbedded build system to create a commercially supportable build system for embedded Linux. After Intel Corporation acquired OpenedHand, the poky project became the basis for the Yocto Project's build system.

- **Recipe:** A set of instructions for building packages. A recipe describes where you get source code, which patches to apply, how to configure the source, how to compile it and so on. Recipes also describe dependencies for libraries or for other recipes. Recipes represent the logical unit of execution, the software to build, the images to build, and use the `.bb` file extension.
- **Reference Kit:** A working example of a system, which includes a [BSP](#) as well as a [build host](#) and other components, that can work on specific hardware.
- **Source Directory:** This term refers to the directory structure created as a result of creating a local copy of the poky Git repository `git://git.yoctoproject.org/poky` or expanding a released poky tarball.

## Note

Creating a local copy of the poky Git repository is the recommended method for setting up your Source Directory.

Sometimes you might hear the term "poky directory" used to refer to this directory structure.

## Note

The OpenEmbedded build system does not support file or directory names that contain spaces. Be sure that the Source Directory you use does not contain these types of names.

The Source Directory contains BitBake, Documentation, Metadata and other files that all support the Yocto Project. Consequently, you must have the Source Directory in place on your development system in order to do any development using the Yocto Project.

When you create a local copy of the Git repository, you can name the repository anything you like. Throughout much of the documentation, "poky" is used as the name of the top-level folder of the local copy of the poky Git repository. So, for example, cloning the poky Git repository results in a local Git repository whose top-level folder is also named "poky".

While it is not recommended that you use tarball expansion to set up the Source Directory, if you do, the top-level directory name of the Source Directory is derived from the Yocto Project release tarball. For example, downloading and unpacking `poky-zeus-22.0.1.tar.bz2` results in a Source Directory whose root folder is named `poky-zeus-22.0.1`.

It is important to understand the differences between the Source Directory created by unpacking a released tarball as compared to cloning `git://git.yoctoproject.org/poky`. When you unpack a tarball, you have an exact copy of the files based on the time of release - a fixed release point. Any changes you make to your local files in the Source Directory are on top of the release and will remain local only. On the other hand, when you clone the poky Git repository, you have an active development repository with access to the upstream repository's branches and tags. In this case, any local changes you make to the local Source Directory can be later applied to active development branches of the upstream poky Git repository.

For more information on concepts related to Git repositories, branches, and tags, see the ["Repositories, Tags, and Branches"](#) section in the Yocto Project Overview and Concepts Manual.

- **Task:** A unit of execution for BitBake (e.g. [do\\_compile](#), [do\\_fetch](#), [do\\_patch](#), and so forth).

- **Toaster:** A web interface to the Yocto Project's [OpenEmbedded Build System](#). The interface enables you to configure and run your builds. Information about builds is collected and stored in a database. For information on Toaster, see the Toaster User Manual.
- **Upstream:** A reference to source code or repositories that are not local to the development system but located in a master area that is controlled by the maintainer of the source code. For example, in order for a developer to work on a particular piece of code, they need to first get a copy of it from an "upstream" source.

## Chapter 23. Yocto Project Releases and the Stable Release Process¶

[23.1. Major and Minor Release Cadence](#)

[23.2. Major Release Codenames](#)

[23.3. Stable Release Process](#)

[23.4. Testing and Quality Assurance](#)

The Yocto Project release process is predictable and consists of both major and minor (point) releases. This brief chapter provides information on how releases are named, their life cycle, and their stability.

### 23.1. Major and Minor Release Cadence¶

The Yocto Project delivers major releases (e.g. 3.0.1) using a six month cadence roughly timed each April and October of the year. Following are examples of some major YP releases with their codenames also shown. See the ["Major Release Codenames"](#) section for information on codenames used with major releases.

```
2.2 (Morty)
2.1 (Krogoth)
2.0 (Jethro)
```

While the cadence is never perfect, this timescale facilitates regular releases that have strong QA cycles while not overwhelming users with too many new releases. The cadence is predictable and avoids many major holidays in various geographies.

The Yocto project delivers minor (point) releases on an unscheduled basis and are usually driven by the accumulation of enough significant fixes or enhancements to the associated major release. Following are some example past point releases:

```
2.1.1
2.1.2
2.2.1
```

The point release indicates a point in the major release branch where a full QA cycle and release process validates the content of the new branch.

#### Note

Realize that there can be patches merged onto the stable release branches as and when they become available.

### 23.2. Major Release Codenames¶

Each major release receives a codename that identifies the release in the [Yocto Project Source Repositories](#). The concept is that branches of [Metadata](#) with the same codename are likely to be compatible and thus work together.

#### Note

Codenames are associated with major releases because a Yocto Project release number (e.g. 3.0.1) could conflict with a given layer or company versioning scheme. Codenames are unique, interesting, and easily identifiable.

Releases are given a nominal release version as well but the codename is used in repositories for this reason. You can find information on Yocto Project releases and codenames at <https://wiki.yoctoproject.org/wiki/Releases>.

### 23.3. Stable Release Process¶

Once released, the release enters the stable release process at which time a person is assigned as the maintainer for that stable release. This maintainer monitors activity for the release by investigating and handling nominated patches and backport activity. Only fixes and enhancements that have first been applied on the "master" branch (i.e. the current, in-development branch) are considered for backporting to a stable release.

#### Note

The current Yocto Project policy regarding backporting is to consider bug fixes and security fixes only. Policy dictates that features are not backported to a stable release. This policy means generic recipe version upgrades are unlikely to be accepted for backporting. The exception to this policy occurs when a strong reason exists such as the fix happens to also be the preferred upstream approach.

Stable release branches have strong maintenance for about a year after their initial release. Should significant issues be found for any release regardless of its age, fixes could be backported to older releases. For issues that are not backported given an older release, Community LTS trees and branches exist where community members share patches for older releases. However, these types of patches do not go through the same release process as do point releases. You can find more information about stable branch maintenance at [https://wiki.yoctoproject.org/wiki/Stable\\_branch\\_maintenance](https://wiki.yoctoproject.org/wiki/Stable_branch_maintenance).

## 23.4. Testing and Quality Assurance¶

Part of the Yocto Project development and release process is quality assurance through the execution of test strategies. Test strategies provide the Yocto Project team a way to ensure a release is validated. Additionally, because the test strategies are visible to you as a developer, you can validate your projects. This section overviews the available test infrastructure used in the Yocto Project. For information on how to run available tests on your projects, see the "[Performing Automated Runtime Testing](#)" section in the Yocto Project Development Tasks Manual.

The QA/testing infrastructure is woven into the project to the point where core developers take some of it for granted. The infrastructure consists of the following pieces:

- `bitbake-selftest`: A standalone command that runs unit tests on key pieces of BitBake and its fetchers.
- `sanity.bbclass`: This automatically included class checks the build environment for missing tools (e.g. `gcc`) or common misconfigurations such as `MACHINE` set incorrectly.
- `insane.bbclass`: This class checks the generated output from builds for sanity. For example, if building for an ARM target, did the build produce ARM binaries. If, for example, the build produced PPC binaries then there is a problem.
- `testimage.bbclass`: This class performs runtime testing of images after they are built. The tests are usually used with `QEMU` to boot the images and check the combined runtime result boot operation and functions. However, the test can also use the IP address of a machine to test.
- `ptest`: Runs tests against packages produced during the build for a given piece of software. The test allows the packages to be run within a target image.
- `oe-selftest`: Tests combination BitBake invocations. These tests operate outside the OpenEmbedded build system itself. The `oe-selftest` can run all tests by default or can run selected tests or test suites.

### Note

Running `oe-selftest` requires host packages beyond the "Essential" grouping. See the "[Required Packages for the Build Host](#)" section for more information.

Originally, much of this testing was done manually. However, significant effort has been made to automate the tests so that more people can use them and the Yocto Project development team can run them faster and more efficiently.

The Yocto Project's main Autobuilder ([autobuilder.yoctoproject.org](http://autobuilder.yoctoproject.org)) publicly tests each Yocto Project release's code in the `OE-Core`, Poky, and BitBake repositories. The testing occurs for both the current state of the "master" branch and also for submitted patches. Testing for submitted patches usually occurs in the "ross/mut" branch in the `poky-contrib` repository (i.e. the master-under-test branch) or in the "master-next" branch in the `poky` repository.

### Note

You can find all these branches in the Yocto Project [Source Repositories](#).

Testing within these public branches ensures in a publicly visible way that all of the main supposed architectures and recipes in OE-Core successfully build and behave properly.

Various features such as `multilib`, sub architectures (e.g. `x32`, `poky-tiny`, `musl`, `no-x11` and so forth), `bitbake-selftest`, and `oe-selftest` are tested as part of the QA process of a release. Complete testing and validation for a release takes the Autobuilder workers several hours.

### Note

The Autobuilder workers are non-homogeneous, which means regular testing across a variety of Linux distributions occurs. The Autobuilder is limited to only testing QEMU-based setups and not real hardware.

Finally, in addition to the Autobuilder's tests, the Yocto Project QA team also performs testing on a variety of platforms, which includes actual hardware, to ensure expected results.

## Chapter 24. Migrating to a Newer Yocto Project Release¶

### 24.1. General Migration Considerations

### 24.2. Moving to the Yocto Project 1.3 Release

#### 24.2.1. Local Configuration

#### 24.2.2. Recipes

#### 24.2.3. Linux Kernel Naming

### 24.3. Moving to the Yocto Project 1.4 Release

#### 24.3.1. BitBake

#### 24.3.2. Build Behavior

#### 24.3.3. Proxies and Fetching Source

- [24.3.4. Custom Interfaces File \(netbase change\).](#)
- [24.3.5. Remote Debugging](#)
- [24.3.6. Variables](#)
- [24.3.7. Target Package Management with RPM](#)
- [24.3.8. Recipes Moved](#)
- [24.3.9. Removals and Renames](#)
- [24.4. Moving to the Yocto Project 1.5 Release](#)
  - [24.4.1. Host Dependency Changes](#)
  - [24.4.2. atom-pc Board Support Package \(BSP\).](#)
  - [24.4.3. BitBake](#)
  - [24.4.4. QA Warnings](#)
  - [24.4.5. Directory Layout Changes](#)
  - [24.4.6. Shortened Git SRCREV Values](#)
  - [24.4.7. IMAGE\\_FEATURES](#)
  - [24.4.8. /run](#)
  - [24.4.9. Removal of Package Manager Database Within Image Recipes](#)
  - [24.4.10. Images Now Rebuild Only on Changes Instead of Every Time](#)
  - [24.4.11. Task Recipes](#)
  - [24.4.12. BusyBox](#)
  - [24.4.13. Automated Image Testing](#)
  - [24.4.14. Build History](#)
  - [24.4.15. udev](#)
  - [24.4.16. Removed and Renamed Recipes](#)
  - [24.4.17. Other Changes](#)
- [24.5. Moving to the Yocto Project 1.6 Release](#)
  - [24.5.1. archiver Class](#)
  - [24.5.2. Packaging Changes](#)
  - [24.5.3. BitBake](#)
  - [24.5.4. Changes to Variables](#)
  - [24.5.5. Package Test \(ptest\).](#)
  - [24.5.6. Build Changes](#)
  - [24.5.7. qemu-native](#)
  - [24.5.8. core-image-basic](#)
  - [24.5.9. Licensing](#)
  - [24.5.10. CFLAGS Options](#)
  - [24.5.11. Custom Image Output Types](#)
  - [24.5.12. Tasks](#)
  - [24.5.13. update-alternative Provider](#)
  - [24.5.14. virtclass Overrides](#)
  - [24.5.15. Removed and Renamed Recipes](#)
  - [24.5.16. Removed Classes](#)
  - [24.5.17. Reference Board Support Packages \(BSPs\).](#)
- [24.6. Moving to the Yocto Project 1.7 Release](#)
  - [24.6.1. Changes to Setting QEMU PACKAGECONFIG Options in local.conf](#)
  - [24.6.2. Minimum Git version](#)
  - [24.6.3. Autotools Class Changes](#)
  - [24.6.4. Binary Configuration Scripts Disabled](#)
  - [24.6.5. eglibc 2.19 Replaced with glibc 2.20](#)
  - [24.6.6. Kernel Module Autoloading](#)
  - [24.6.7. QA Check Changes](#)
  - [24.6.8. Removed Recipes](#)
  - [24.6.9. Miscellaneous Changes](#)
- [24.7. Moving to the Yocto Project 1.8 Release](#)
  - [24.7.1. Removed Recipes](#)
  - [24.7.2. BlueZ 4.x / 5.x Selection](#)
  - [24.7.3. Kernel Build Changes](#)
  - [24.7.4. SSL 3.0 is Now Disabled in OpenSSL](#)
  - [24.7.5. Default Sysroot Poisoning](#)
  - [24.7.6. Rebuild Improvements](#)
  - [24.7.7. QA Check and Validation Changes](#)
  - [24.7.8. Miscellaneous Changes](#)
- [24.8. Moving to the Yocto Project 2.0 Release](#)
  - [24.8.1. GCC 5](#)
  - [24.8.2. Gstreamer 0.10 Removed](#)
  - [24.8.3. Removed Recipes](#)
  - [24.8.4. BitBake datastore improvements](#)
  - [24.8.5. Shell Message Function Changes](#)
  - [24.8.6. Extra Development/Debug Package Cleanup](#)
  - [24.8.7. Recipe Maintenance Tracking Data Moved to OE-Core](#)
  - [24.8.8. Automatic Stale Sysroot File Cleanup](#)
  - [24.8.9. linux-yocto Kernel Metadata Repository Now Split from Source](#)
  - [24.8.10. Additional QA checks](#)
  - [24.8.11. Miscellaneous Changes](#)
- [24.9. Moving to the Yocto Project 2.1 Release](#)
  - [24.9.1. Variable Expansion in Python Functions](#)
  - [24.9.2. Overrides Must Now be Lower-Case](#)
  - [24.9.3. Expand Parameter to getVar\(.\) and getVarFlag\(.\) is Now Mandatory](#)
  - [24.9.4. Makefile Environment Changes](#)
  - [24.9.5. libexecdir Reverted to \\${prefix}/libexec](#)
  - [24.9.6. ac\\_cv\\_sizeof\\_off\\_t is No Longer Cached in Site Files](#)
  - [24.9.7. Image Generation is Now Split Out from Filesystem Generation](#)
  - [24.9.8. Removed Recipes](#)
  - [24.9.9. Class Changes](#)
  - [24.9.10. Build System User Interface Changes](#)
  - [24.9.11. ADT Removed](#)
  - [24.9.12. Poky Reference Distribution Changes](#)
  - [24.9.13. Packaging Changes](#)
  - [24.9.14. Tuning File Changes](#)
  - [24.9.15. Supporting GObject Introspection](#)
  - [24.9.16. Miscellaneous Changes](#)
- [24.10. Moving to the Yocto Project 2.2 Release](#)
  - [24.10.1. Minimum Kernel Version](#)
  - [24.10.2. Staging Directories in Sysroot Has Been Simplified](#)



- [24.10.3. Removal of Old Images and Other Files in `tmp/deploy` Now Enabled](#)
- [24.10.4. Python Changes](#)
- [24.10.5. `uClibc` Replaced by `musl`](#)
- [24.10.6. `\$\(B\)` No Longer Default Working Directory for Tasks](#)
- [24.10.7. `runqemu` Ported to Python](#)
- [24.10.8. Default Linker Hash Style Changed](#)
- [24.10.9. `KERNEL\_IMAGE\_BASE\_NAME` no Longer Uses `KERNEL\_IMAGETYPE`](#)
- [24.10.10. BitBake Changes](#)
- [24.10.11. Swabber has Been Removed](#)
- [24.10.12. Removed Recipes](#)
- [24.10.13. Removed Classes](#)
- [24.10.14. Minor Packaging Changes](#)
- [24.10.15. Miscellaneous Changes](#)
- [24.11. Moving to the Yocto Project 2.3 Release](#)
  - [24.11.1. Recipe-specific Sysroots](#)
  - [24.11.2. `PATH` Variable](#)
  - [24.11.3. Changes to Scripts](#)
  - [24.11.4. Changes to Functions](#)
  - [24.11.5. BitBake Changes](#)
  - [24.11.6. Absolute Symbolic Links](#)
  - [24.11.7. GPLv2 Versions of GPLv3 Recipes Moved](#)
  - [24.11.8. Package Management Changes](#)
  - [24.11.9. Removed Recipes](#)
  - [24.11.10. Wic Changes](#)
  - [24.11.11. QA Changes](#)
  - [24.11.12. Miscellaneous Changes](#)
- [24.12. Moving to the Yocto Project 2.4 Release](#)
  - [24.12.1. Memory Resident Mode](#)
  - [24.12.2. Packaging Changes](#)
  - [24.12.3. Removed Recipes](#)
  - [24.12.4. Kernel Device Tree Move](#)
  - [24.12.5. Package QA Changes](#)
  - [24.12.6. `README` File Changes](#)
  - [24.12.7. Miscellaneous Changes](#)
- [24.13. Moving to the Yocto Project 2.5 Release](#)
  - [24.13.1. Packaging Changes](#)
  - [24.13.2. Removed Recipes](#)
  - [24.13.3. Scripts and Tools Changes](#)
  - [24.13.4. BitBake Changes](#)
  - [24.13.5. Python and Python 3 Changes](#)
  - [24.13.6. Miscellaneous Changes](#)
- [24.14. Moving to the Yocto Project 2.6 Release](#)
  - [24.14.1. GCC 8.2 is Now Used by Default](#)
  - [24.14.2. Removed Recipes](#)
  - [24.14.3. Packaging Changes](#)
  - [24.14.4. XOrg Protocol dependencies](#)
  - [24.14.5. `distutils` and `distutils3` Now Prevent Fetching Dependencies During the `do\_configure` Task](#)
  - [24.14.6. `linux-yocto` Configuration Audit Issues Now Correctly Reported](#)
  - [24.14.7. Image/Kernel Artifact Naming Changes](#)
  - [24.14.8. `SERIAL\_CONSOLE` Deprecated](#)
  - [24.14.9. Configure Script Reports Unknown Options as Errors](#)
  - [24.14.10. Override Changes](#)
  - [24.14.11. `systemd` Configuration is Now Split Into `systemd-conf`](#)
  - [24.14.12. Automatic Testing Changes](#)
  - [24.14.13. OpenSSL Changes](#)
  - [24.14.14. BitBake Changes](#)
  - [24.14.15. Security Changes](#)
  - [24.14.16. Post Installation Changes](#)
  - [24.14.17. Python 3 Profile-Guided Optimization](#)
  - [24.14.18. Miscellaneous Changes](#)
- [24.15. Moving to the Yocto Project 2.7 Release](#)
  - [24.15.1. BitBake Changes](#)
  - [24.15.2. Eclipse™ Support Removed](#)
  - [24.15.3. `qemu-native` Splits the System and User-Mode Parts](#)
  - [24.15.4. The `upstream-tracking.inc` File Has Been Removed](#)
  - [24.15.5. The `DISTRO\_FEATURES\_LIBC` Variable Has Been Removed](#)
  - [24.15.6. License Value Corrections](#)
  - [24.15.7. Packaging Changes](#)
  - [24.15.8. Removed Recipes](#)
  - [24.15.9. Removed Classes](#)
  - [24.15.10. Miscellaneous Changes](#)
- [24.16. Moving to the Yocto Project 3.0 Release](#)
  - [24.16.1. Init System Selection](#)
  - [24.16.2. LSB Support Removed](#)
  - [24.16.3. Removed Recipes](#)
  - [24.16.4. Packaging Changes](#)
  - [24.16.5. CVE Checking](#)
  - [24.16.6. Bitbake Changes](#)
  - [24.16.7. Sanity Checks](#)
  - [24.16.8. Miscellaneous Changes](#)

This chapter provides information you can use to migrate work to a newer Yocto Project release. You can find the same information in the release notes for a given release.

## 24.1. General Migration Considerations¶

Some considerations are not tied to a specific Yocto Project release. This section presents information you should consider when migrating to any new Yocto Project release.

- **Dealing with Customized Recipes:** Issues could arise if you take older recipes that contain customizations and simply copy them forward expecting them to work after you migrate to new Yocto Project metadata. For example, suppose you have a recipe in your

layer that is a customized version of a core recipe copied from the earlier release, rather than through the use of an append file. When you migrate to a newer version of Yocto Project, the metadata (e.g. perhaps an include file used by the recipe) could have changed in a way that would break the build. Say, for example, a function is removed from an include file and the customized recipe tries to call that function.

You could "forward-port" all your customizations in your recipe so that everything works for the new release. However, this is not the optimal solution as you would have to repeat this process with each new release if changes occur that give rise to problems.

The better solution (where practical) is to use append files (`*.bbappend`) to capture any customizations you want to make to a recipe. Doing so, isolates your changes from the main recipe making them much more manageable. However, sometimes it is not practical to use an append file. A good example of this is when introducing a newer or older version of a recipe in another layer.

- **Updating Append Files:** Since append files generally only contain your customizations, they often do not need to be adjusted for new releases. However, if the `.bbappend` file is specific to a particular version of the recipe (i.e. its name does not use the % wildcard) and the version of the recipe to which it is appending has changed, then you will at a minimum need to rename the append file to match the name of the recipe file. A mismatch between an append file and its corresponding recipe file (`.bb`) will trigger an error during parsing.

Depending on the type of customization the append file applies, other incompatibilities might occur when you upgrade. For example, if your append file applies a patch and the recipe to which it is appending is updated to a newer version, the patch might no longer apply. If this is the case and assuming the patch is still needed, you must modify the patch file so that it does apply.

## 24.2. Moving to the Yocto Project 1.3 Release¶

This section provides migration information for moving to the Yocto Project 1.3 Release from the prior release.

### 24.2.1. Local Configuration¶

Differences include changes for `SSTATE_MIRRORS` and `bblayers.conf`.

#### 24.2.1.1. SSTATE\_MIRRORS¶

The shared state cache (sstate-cache), as pointed to by `SSTATE_DIR`, by default now has two-character subdirectories to prevent issues arising from too many files in the same directory. Also, native sstate-cache packages, which are built to run on the host system, will go into a subdirectory named using the distro ID string. If you copy the newly structured sstate-cache to a mirror location (either local or remote) and then point to it in `SSTATE_MIRRORS`, you need to append "PATH" to the end of the mirror URL so that the path used by BitBake before the mirror substitution is appended to the path used to access the mirror. Here is an example:

```
SSTATE_MIRRORS = "file://.* http://someserver.tld/share/sstate/PATH"
```

#### 24.2.1.2. bblayers.conf¶

The `meta-yocto` layer consists of two parts that correspond to the Poky reference distribution and the reference hardware Board Support Packages (BSPs), respectively: `meta-yocto` and `meta-yocto-bsp`. When running BitBake for the first time after upgrading, your `conf/bblayers.conf` file will be updated to handle this change and you will be asked to re-run or restart for the changes to take effect.

### 24.2.2. Recipes¶

Differences include changes for the following:

- Python function whitespace
- `proto=` in `SRC_URI`
- `nativesdk`
- Task recipes
- `IMAGE_FEATURES`
- Removed recipes

#### 24.2.2.1. Python Function Whitespace¶

All Python functions must now use four spaces for indentation. Previously, an inconsistent mix of spaces and tabs existed, which made extending these functions using `_append` or `_prepend` complicated given that Python treats whitespace as syntactically significant. If you are defining or extending any Python functions (e.g. `populate_packages`, `do_unpack`, `do_patch` and so forth) in custom recipes or classes, you need to ensure you are using consistent four-space indentation.

#### 24.2.2.2. proto= in SRC\_URI¶

Any use of `proto=` in `SRC_URI` needs to be changed to `protocol=`. In particular, this applies to the following URIs:

- `svn://`
- `bzr://`
- `hg://`
- `osc://`

Other URIs were already using `protocol=`. This change improves consistency.

#### 24.2.2.3. nativesdk¶

The suffix `nativesdk` is now implemented as a prefix, which simplifies a lot of the packaging code for `nativesdk` recipes. All custom `nativesdk` recipes, which are relocatable packages that are native to `SDK_ARCH`, and any references need to be updated to use `nativesdk-*` instead of `*-nativesdk`.

#### 24.2.2.4. Task Recipes¶

"Task" recipes are now known as "Package groups" and have been renamed from `task-*.bb` to `packagegroup-*.bb`. Existing references to the previous `task-*` names should work in most cases as there is an automatic upgrade path for most packages. However, you should update references in your own recipes and configurations as they could be removed in future releases. You should also rename any custom `task-*` recipes to `packagegroup-*`, and change them to inherit `packagegroup` instead of `task`, as well as taking the opportunity to remove anything now handled by `packagegroup.bbclass`, such as providing `-dev` and `-dbg` packages, setting `LIC_FILES_CHKSUM`, and so forth. See the "[packagegroup.bbclass](#)" section for further details.

#### 24.2.2.5. IMAGE\_FEATURES¶

Image recipes that previously included "apps-console-core" in `IMAGE_FEATURES` should now include "splash" instead to enable the boot-up splash screen. Retaining "apps-console-core" will still include the splash screen but generates a warning. The "apps-x11-core" and "apps-x11-games" `IMAGE_FEATURES` features have been removed.

#### 24.2.2.6. Removed Recipes¶

The following recipes have been removed. For most of them, it is unlikely that you would have any references to them in your own [Metadata](#). However, you should check your metadata against this list to be sure:

- **`libx11-trim`**: Replaced by `libx11`, which has a negligible size difference with modern Xorg.
- **`xserver-xorg-lite`**: Use `xserver-xorg`, which has a negligible size difference when DRI and GLX modules are not installed.
- **`xserver-kdrive`**: Effectively unmaintained for many years.
- **`mesa-xlib`**: No longer serves any purpose.
- **`galago`**: Replaced by `telepathy`.
- **`gail`**: Functionality was integrated into GTK+ 2.13.
- **`eggdbus`**: No longer needed.
- **`gcc-*-intermediate`**: The build has been restructured to avoid the need for this step.
- **`libgsmd`**: Unmaintained for many years. Functionality now provided by `ofono` instead.
- **`contacts`, `dates`, `tasks`, `eds-tools`**: Largely unmaintained PIM application suite. It has been moved to `meta-gnome` in `meta-openembedded`.

In addition to the previously listed changes, the `meta-demoapps` directory has also been removed because the recipes in it were not being maintained and many had become obsolete or broken. Additionally, these recipes were not parsed in the default configuration. Many of these recipes are already provided in an updated and maintained form within the OpenEmbedded community layers such as `meta-oe` and `meta-gnome`. For the remainder, you can now find them in the `meta-extras` repository, which is in the Yocto Project [Source Repositories](#).

#### 24.2.3. Linux Kernel Naming¶

The naming scheme for kernel output binaries has been changed to now include `PE` as part of the filename:

```
KERNEL_IMAGE_BASE_NAME ?= "${KERNEL_IMAGETYPE}-${PE}-${PV}-${PR}-${MACHINE}-${DATETIME}"
```

Because the `PE` variable is not set by default, these binary files could result with names that include two dash characters. Here is an example:

```
bzImage--3.10.9+git0+cd502a8814_7144bcc4b8-r0-qemux86-64-20130830085431.bin
```

### 24.3. Moving to the Yocto Project 1.4 Release¶

This section provides migration information for moving to the Yocto Project 1.4 Release from the prior release.

#### 24.3.1. BitBake¶

Differences include the following:

- ***Comment Continuation***: If a comment ends with a line continuation (`\`) character, then the next line must also be a comment. Any instance where this is not the case, now triggers a warning. You must either remove the continuation character, or be sure the next line is a comment.
- ***Package Name Overrides***: The runtime package specific variables `RDEPENDS`, `RRECOMMENDS`, `RSUGGESTS`, `RPROVIDES`, `RCONFLICTS`, `RREPLACES`, `FILES`, `ALLOW_EMPTY`, and the `pre`, `post`, `install`, and `uninstall` script functions `pkg_preinst`, `pkg_postinst`, `pkg_prerm`, and `pkg_postrm` should always have a package name override. For example, use `RDEPENDS_${PN}` for the main package instead of `RDEPENDS`. BitBake uses more strict checks when it parses recipes.

#### 24.3.2. Build Behavior¶

Differences include the following:

- **Shared State Code:** The shared state code has been optimized to avoid running unnecessary tasks. For example, the following no longer populates the target sysroot since that is not necessary:

```
$ bitbake -c rootfs some-image
```

Instead, the system just needs to extract the output package contents, re-create the packages, and construct the root filesystem. This change is unlikely to cause any problems unless you have missing declared dependencies.

- **Scanning Directory Names:** When scanning for files in `SRC_URI`, the build system now uses `FILESOVERRIDES` instead of `OVERRIDES` for the directory names. In general, the values previously in `OVERRIDES` are now in `FILESOVERRIDES` as well. However, if you relied upon an additional value you previously added to `OVERRIDES`, you might now need to add it to `FILESOVERRIDES` unless you are already adding it through the `MACHINEOVERRIDES` or `DISTROOVERRIDES` variables, as appropriate. For more related changes, see the "[Variables](#)" section.

### 24.3.3. Proxies and Fetching Source¶

A new `oe-git-proxy` script has been added to replace previous methods of handling proxies and fetching source from Git. See the `meta-yocto/conf/site.conf.sample` file for information on how to use this script.

### 24.3.4. Custom Interfaces File (netbase change)¶

If you have created your own custom `etc/network/interfaces` file by creating an append file for the `netbase` recipe, you now need to create an append file for the `init-ifupdown` recipe instead, which you can find in the [Source Directory](#) at `meta/recipes-core/init-ifupdown`. For information on how to use append files, see the "[Using .bbappend Files](#)" section in the Yocto Project Development Tasks Manual.

### 24.3.5. Remote Debugging¶

Support for remote debugging with the Eclipse IDE is now separated into an image feature (`eclipse-debug`) that corresponds to the `packagegroup-core-eclipse-debug` package group. Previously, the debugging feature was included through the `tools-debug` image feature, which corresponds to the `packagegroup-core-tools-debug` package group.

### 24.3.6. Variables¶

The following variables have changed:

- **`SANITY_TESTED_DISTROS`:** This variable now uses a distribution ID, which is composed of the host distributor ID followed by the release. Previously, `SANITY_TESTED_DISTROS` was composed of the description field. For example, "Ubuntu 12.10" becomes "Ubuntu-12.10". You do not need to worry about this change if you are not specifically setting this variable, or if you are specifically setting it to "".
- **`SRC_URI`:** The `${PN}`, `${PF}`, `${P}`, and `FILE_DIRNAME` directories have been dropped from the default value of the `FILESPATH` variable, which is used as the search path for finding files referred to in `SRC_URI`. If you have a recipe that relied upon these directories, which would be unusual, then you will need to add the appropriate paths within the recipe or, alternatively, rearrange the files. The most common locations are still covered by `${BP}`, `${BPN}`, and "files", which all remain in the default value of `FILESPATH`.

### 24.3.7. Target Package Management with RPM¶

If runtime package management is enabled and the RPM backend is selected, Smart is now installed for package download, dependency resolution, and upgrades instead of Zypper. For more information on how to use Smart, run the following command on the target:

```
smart --help
```

### 24.3.8. Recipes Moved¶

The following recipes were moved from their previous locations because they are no longer used by anything in the OpenEmbedded-Core:

- **`clutter-box2d`:** Now resides in the `meta-oe` layer.
- **`evolution-data-server`:** Now resides in the `meta-gnome` layer.
- **`gthumb`:** Now resides in the `meta-gnome` layer.
- **`gtkhtml2`:** Now resides in the `meta-oe` layer.
- **`gupnp`:** Now resides in the `meta-multimedia` layer.
- **`gypsy`:** Now resides in the `meta-oe` layer.
- **`libcanberra`:** Now resides in the `meta-gnome` layer.
- **`libgdata`:** Now resides in the `meta-gnome` layer.
- **`libmusicbrainz`:** Now resides in the `meta-multimedia` layer.
- **`metacity`:** Now resides in the `meta-gnome` layer.
- **`polkit`:** Now resides in the `meta-oe` layer.
- **`zeroconf`:** Now resides in the `meta-networking` layer.

### 24.3.9. Removals and Renames¶

The following list shows what has been removed or renamed:

- **evieext:** Removed because it has been removed from `xserver` since 2008.
- **Gtk+ DirectFB:** Removed support because upstream Gtk+ no longer supports it as of version 2.18.
- **libxfontcache / xfontcacheprotos:** Removed because they were removed from the Xorg server in 2008.
- **libxp / libxprintapputil / libxprintutil / printproto:** Removed because the XPrint server was removed from Xorg in 2008.
- **libxtrap / xtrapproto:** Removed because their functionality was broken upstream.
- **linux-yocto 3.0 kernel:** Removed with linux-yocto 3.8 kernel being added. The linux-yocto 3.2 and linux-yocto 3.4 kernels remain as part of the release.
- **lsbsetup:** Removed with functionality now provided by `lsbtest`.
- **matchbox-stroke:** Removed because it was never more than a proof-of-concept.
- **matchbox-wm-2 / matchbox-theme-sato-2:** Removed because they are not maintained. However, `matchbox-wm` and `matchbox-theme-sato` are still provided.
- **mesa-dri:** Renamed to `mesa`.
- **mesa-xlib:** Removed because it was no longer useful.
- **mutter:** Removed because nothing ever uses it and the recipe is very old.
- **orinoco-conf:** Removed because it has become obsolete.
- **update-modules:** Removed because it is no longer used. The kernel module `postinstall` and `postrm` scripts can now do the same task without the use of this script.
- **web:** Removed because it is not maintained. Superseded by `web-webkit`.
- **xf86bigfontproto:** Removed because upstream it has been disabled by default since 2007. Nothing uses `xf86bigfontproto`.
- **xf86rushproto:** Removed because its dependency in `xserver` was spurious and it was removed in 2005.
- **zypper / libzypp / sat-solver:** Removed and been functionally replaced with `Smart` (`python-smartpm`) when RPM packaging is used and package management is enabled on the target.

## 24.4. Moving to the Yocto Project 1.5 Release¶

This section provides migration information for moving to the Yocto Project 1.5 Release from the prior release.

### 24.4.1. Host Dependency Changes¶

The OpenEmbedded build system now has some additional requirements on the host system:

- Python 2.7.3+
- Tar 1.24+
- Git 1.7.8+
- Patched version of Make if you are using 3.82. Most distributions that provide Make 3.82 use the patched version.

If the Linux distribution you are using on your build host does not provide packages for these, you can install and use the Buildtools tarball, which provides an SDK-like environment containing them.

For more information on this requirement, see the "[Required Git, tar, and Python Versions](#)" section.

### 24.4.2. atom-pc Board Support Package (BSP)¶

The `atom-pc` hardware reference BSP has been replaced by a `genericx86` BSP. This BSP is not necessarily guaranteed to work on all x86 hardware, but it will run on a wider range of systems than the `atom-pc` did.

#### Note

Additionally, a `genericx86-64` BSP has been added for 64-bit Atom systems.

### 24.4.3. BitBake¶

The following changes have been made that relate to BitBake:

- BitBake now supports a `__remove` operator. The addition of this operator means you will have to rename any items in recipe space (functions, variables) whose names currently contain `__remove__` or end with `__remove` to avoid unexpected behavior.
- BitBake's global method pool has been removed. This method is not particularly useful and led to clashes between recipes containing functions that had the same name.
- The "none" server backend has been removed. The "process" server backend has been serving well as the default for a long time now.
- The `bitbake-runtask` script has been removed.

- `_${P}` and `_${PF}` are no longer added to `PROVIDES` by default in `bitbake.conf`. These version-specific `PROVIDES` items were seldom used. Attempting to use them could result in two versions being built simultaneously rather than just one version due to the way BitBake resolves dependencies.

#### 24.4.4. QA Warnings¶

The following changes have been made to the package QA checks:

- If you have customized `ERROR_QA` or `WARN_QA` values in your configuration, check that they contain all of the issues that you wish to be reported. Previous Yocto Project versions contained a bug that meant that any item not mentioned in `ERROR_QA` or `WARN_QA` would be treated as a warning. Consequently, several important items were not already in the default value of `WARN_QA`. All of the possible QA checks are now documented in the "[insane.bbclass](#)" section.
- An additional QA check has been added to check if `/usr/share/info/dir` is being installed. Your recipe should delete this file within `do_install` if "make install" is installing it.
- If you are using the `buildhistory` class, the check for the package version going backwards is now controlled using a standard QA check. Thus, if you have customized your `ERROR_QA` or `WARN_QA` values and still wish to have this check performed, you should add "version-going-backwards" to your value for one or the other variables depending on how you wish it to be handled. See the documented QA checks in the "[insane.bbclass](#)" section.

#### 24.4.5. Directory Layout Changes¶

The following directory changes exist:

- Output SDK installer files are now named to include the image name and tuning architecture through the `SDK_NAME` variable.
- Images and related files are now installed into a directory that is specific to the machine, instead of a parent directory containing output files for multiple machines. The `DEPLOY_DIR_IMAGE` variable continues to point to the directory containing images for the current `MACHINE` and should be used anywhere there is a need to refer to this directory. The `runqemu` script now uses this variable to find images and kernel binaries and will use BitBake to determine the directory. Alternatively, you can set the `DEPLOY_DIR_IMAGE` variable in the external environment.
- When `buildhistory` is enabled, its output is now written under the `Build Directory` rather than `TMPDIR`. Doing so makes it easier to delete `TMPDIR` and preserve the build history. Additionally, data for produced SDKs is now split by `IMAGE_NAME`.
- The `pkgdata` directory produced as part of the packaging process has been collapsed into a single machine-specific directory. This directory is located under `sysroots` and uses a machine-specific name (i.e. `tmp/sysroots/machine/pkgdata`).

#### 24.4.6. Shortened Git SRCREV Values¶

BitBake will now shorten revisions from Git repositories from the normal 40 characters down to 10 characters within `SRCPV` for improved usability in path and file names. This change should be safe within contexts where these revisions are used because the chances of spatially close collisions is very low. Distant collisions are not a major issue in the way the values are used.

#### 24.4.7. IMAGE\_FEATURES¶

The following changes have been made that relate to `IMAGE_FEATURES`:

- The value of `IMAGE_FEATURES` is now validated to ensure invalid feature items are not added. Some users mistakenly add package names to this variable instead of using `IMAGE_INSTALL` in order to have the package added to the image, which does not work. This change is intended to catch those kinds of situations. Valid `IMAGE_FEATURES` are drawn from `PACKAGE_GROUP` definitions, `COMPLEMENTARY_GLOB` and a new "validitems" varflag on `IMAGE_FEATURES`. The "validitems" varflag change allows additional features to be added if they are not provided using the previous two mechanisms.
- The previously deprecated "apps-console-core" `IMAGE_FEATURES` item is no longer supported. Add "splash" to `IMAGE_FEATURES` if you wish to have the splash screen enabled, since this is all that apps-console-core was doing.

#### 24.4.8. /run¶

The `/run` directory from the Filesystem Hierarchy Standard 3.0 has been introduced. You can find some of the implications for this change [here](#). The change also means that recipes that install files to `/var/run` must be changed. You can find a guide on how to make these changes [here](#).

#### 24.4.9. Removal of Package Manager Database Within Image Recipes¶

The image `core-image-minimal` no longer adds `remove_packaging_data_files` to `ROOTFS_POSTPROCESS_COMMAND`. This addition is now handled automatically when "package-management" is not in `IMAGE_FEATURES`. If you have custom image recipes that make this addition, you should remove the lines, as they are not needed and might interfere with correct operation of postinstall scripts.

#### 24.4.10. Images Now Rebuild Only on Changes Instead of Every Time¶

The `do_rootfs` and other related image construction tasks are no longer marked as "nostamp". Consequently, they will only be re-executed when their inputs have changed. Previous versions of the OpenEmbedded build system always rebuilt the image when requested rather than necessary.

#### 24.4.11. Task Recipes¶

The previously deprecated `task.bbclass` has now been dropped. For recipes that previously inherited from this class, you should rename them from `task-*` to `packagegroup-*` and inherit `packagegroup` instead.

For more information, see the "[packagegroup.bbclass](#)" section.

#### 24.4.12. BusyBox¶

By default, we now split BusyBox into two binaries: one that is `suid root` for those components that need it, and another for the rest of the components. Splitting BusyBox allows for optimization that eliminates the `tinylogin` recipe as recommended by upstream. You can disable this split by setting `BUSYBOX_SPLIT_SUID` to "0".

#### 24.4.13. Automated Image Testing¶

A new automated image testing framework has been added through the `testimage.bbclass` class. This framework replaces the older `imagetest-qemu` framework.

You can learn more about performing automated image tests in the "[Performing Automated Runtime Testing](#)" section in the Yocto Project Development Tasks Manual.

#### 24.4.14. Build History¶

Following are changes to Build History:

- Installed package sizes: `installed-package-sizes.txt` for an image now records the size of the files installed by each package instead of the size of each compressed package archive file.
- The dependency graphs (`depends*.dot`) now use the actual package names instead of replacing dashes, dots and plus signs with underscores.
- The `buildhistory-diff` and `buildhistory-collect-screvs` utilities have improved command-line handling. Use the `--help` option for each utility for more information on the new syntax.

For more information on Build History, see the "[Maintaining Build Output Quality](#)" section in the Yocto Project Development Tasks Manual.

#### 24.4.15. udev¶

Following are changes to udev:

- `udev` no longer brings in `udev-extraconf` automatically through `RRECOMMENDS`, since this was originally intended to be optional. If you need the extra rules, then add `udev-extraconf` to your image.
- `udev` no longer brings in `pciutils-ids` or `usbutils-ids` through `RRECOMMENDS`. These are not needed by `udev` itself and removing them saves around 350KB.

#### 24.4.16. Removed and Renamed Recipes¶

- The `linux-yocto 3.2` kernel has been removed.
- `libtool-nativesdk` has been renamed to `nativesdk-libtool`.
- `tinylogin` has been removed. It has been replaced by a `suid` portion of Busybox. See the "[BusyBox](#)" section for more information.
- `external-python-tarball` has been renamed to `buildtools-tarball`.
- `web-webkit` has been removed. It has been functionally replaced by `midori`.
- `imake` has been removed. It is no longer needed by any other recipe.
- `transfig-native` has been removed. It is no longer needed by any other recipe.
- `anjuta-remote-run` has been removed. Anjuta IDE integration has not been officially supported for several releases.

#### 24.4.17. Other Changes¶

Following is a list of short entries describing other changes:

- `run-postinsts`: Make this generic.
- `base-files`: Remove the unnecessary `media/xxx` directories.
- `alsa-state`: Provide an empty `asound.conf` by default.
- `classes/image`: Ensure `BAD_RECOMMENDATIONS` supports pre-renamed package names.
- `classes/rootfs_rpm`: Implement `BAD_RECOMMENDATIONS` for RPM.
- `systemd`: Remove `systemd_unitdir` if `systemd` is not in `DISTRO_FEATURES`.
- `systemd`: Remove `init.d` dir if `systemd` unit file is present and `sysvinit` is not a distro feature.
- `libpam`: Deny all services for the `OTHER` entries.
- `image.bbclass`: Move `runtime_mapping_rename` to avoid conflict with `multilib`. See [YOCTO #4993](#) in Bugzilla for more information.
- `linux-dtb`: Use kernel build system to generate the `dtb` files.
- `kern-tools`: Switch from `guilt` to new `kgit-s2q` tool.

### 24.5. Moving to the Yocto Project 1.6 Release¶

This section provides migration information for moving to the Yocto Project 1.6 Release from the prior release.



### 24.5.1. archiver Class¶

The `archiver` class has been rewritten and its configuration has been simplified. For more details on the source archiver, see the "[Maintaining Open Source License Compliance During Your Product's Lifecycle](#)" section in the Yocto Project Development Tasks Manual.

### 24.5.2. Packaging Changes¶

The following packaging changes have been made:

- The `binutils` recipe no longer produces a `binutils-symlinks` package. `update-alternatives` is now used to handle the preferred `binutils` variant on the target instead.
- The `tc` (traffic control) utilities have been split out of the main `iproute2` package and put into the `iproute2-tc` package.
- The `gtk-engines` schemas have been moved to a dedicated `gtk-engines-schemas` package.
- The `armv7a` with thumb package architecture suffix has changed. The suffix for these packages with the thumb optimization enabled is "t2" as it should be. Use of this suffix was not the case in the 1.5 release. Architecture names will change within package feeds as a result.

### 24.5.3. BitBake¶

The following changes have been made to [BitBake](#).

#### 24.5.3.1. Matching Branch Requirement for Git Fetching¶

When fetching source from a Git repository using `SRC_URI`, BitBake will now validate the `SRCREV` value against the branch. You can specify the branch using the following form:

```
SRC_URI = "git://server.name/repository;branch=branchname"
```

If you do not specify a branch, BitBake looks in the default "master" branch.

Alternatively, if you need to bypass this check (e.g. if you are fetching a revision corresponding to a tag that is not on any branch), you can add `;"nobranch=1"` to the end of the URL within `SRC_URI`.

#### 24.5.3.2. Python Definition substitutions¶

BitBake had some previously deprecated Python definitions within its `bb` module removed. You should use their sub-module counterparts instead:

- `bb.MalformedUrl`: Use `bb.fetch.MalformedUrl`.
- `bb.encodeurl`: Use `bb.fetch.encodeurl`.
- `bb.decodeurl`: Use `bb.fetch.decodeurl`.
- `bb.mkdirhier`: Use `bb.utils.mkdirhier`.
- `bb.movefile`: Use `bb.utils.movefile`.
- `bb.copyfile`: Use `bb.utils.copyfile`.
- `bb.which`: Use `bb.utils.which`.
- `bb.vercmp_string`: Use `bb.utils.vercmp_string`.
- `bb.vercmp`: Use `bb.utils.vercmp`.

#### 24.5.3.3. SVK Fetcher¶

The SVK fetcher has been removed from BitBake.

#### 24.5.3.4. Console Output Error Redirection¶

The BitBake console UI will now output errors to `stderr` instead of `stdout`. Consequently, if you are piping or redirecting the output of `bitbake` to somewhere else, and you wish to retain the errors, you will need to add `2>&1` (or something similar) to the end of your `bitbake` command line.

#### 24.5.3.5. task-taskname Overrides¶

`task-taskname` overrides have been adjusted so that tasks whose names contain underscores have the underscores replaced by hyphens for the override so that they now function properly. For example, the task override for `do_populate_sdk` is `task-populate-sdk`.

### 24.5.4. Changes to Variables¶

The following variables have changed. For information on the OpenEmbedded build system variables, see the "[Variables Glossary](#)" Chapter.

#### 24.5.4.1. TMPDIR¶

`TMPDIR` can no longer be on an NFS mount. NFS does not offer full POSIX locking and inode consistency and can cause unexpected issues if used to store `TMPDIR`.

The check for this occurs on startup. If `TMPDIR` is detected on an NFS mount, an error occurs.



#### 24.5.4.2. PRINC¶

The `PRINC` variable has been deprecated and triggers a warning if detected during a build. For `PR` increments on changes, use the `PR` service instead. You can find out more about this service in the "[Working With a PR Service](#)" section in the Yocto Project Development Tasks Manual.

#### 24.5.4.3. IMAGE\_TYPES¶

The `"sum.jffs2"` option for `IMAGE_TYPES` has been replaced by the `"jffs2.sum"` option, which fits the processing order.

#### 24.5.4.4. COPY\_LIC\_MANIFEST¶

The `COPY_LIC_MANIFEST` variable must now be set to `"1"` rather than any value in order to enable it.

#### 24.5.4.5. COPY\_LIC\_DIRS¶

The `COPY_LIC_DIRS` variable must now be set to `"1"` rather than any value in order to enable it.

#### 24.5.4.6. PACKAGE\_GROUP¶

The `PACKAGE_GROUP` variable has been renamed to `FEATURE_PACKAGES` to more accurately reflect its purpose. You can still use `PACKAGE_GROUP` but the OpenEmbedded build system produces a warning message when it encounters the variable.

#### 24.5.4.7. Preprocess and Post Process Command Variable Behavior¶

The following variables now expect a semicolon separated list of functions to call and not arbitrary shell commands:

```
ROOTFS_PREPROCESS_COMMAND
ROOTFS_POSTPROCESS_COMMAND
SDK_POSTPROCESS_COMMAND
POPULATE_SDK_POST_TARGET_COMMAND
POPULATE_SDK_POST_HOST_COMMAND
IMAGE_POSTPROCESS_COMMAND
IMAGE_PREPROCESS_COMMAND
ROOTFS_POSTUNINSTALL_COMMAND
ROOTFS_POSTINSTALL_COMMAND
```

For migration purposes, you can simply wrap shell commands in a shell function and then call the function. Here is an example:

```
my_postprocess_function() {
 echo "hello" > ${IMAGE_ROOTFS}/hello.txt
}
ROOTFS_POSTPROCESS_COMMAND += "my_postprocess_function; "
```

#### 24.5.5. Package Test (ptest)¶

Package Tests (ptest) are built but not installed by default. For information on using Package Tests, see the "[Testing Packages with ptest](#)" section in the Yocto Project Development Tasks Manual. For information on the `ptest.bbclass`, see the "[ptest.bbclass](#)" section.

#### 24.5.6. Build Changes¶

Separate build and source directories have been enabled by default for selected recipes where it is known to work (a whitelist) and for all recipes that inherit the `cmake` class. In future releases the `autotools` class will enable a separate build directory by default as well. Recipes building Autotools-based software that fails to build with a separate build directory should be changed to inherit from the `autotools-brokensep` class instead of the `autotools` or `autotools_staged` classes.

#### 24.5.7. qemu-native¶

`qemu-native` now builds without SDL-based graphical output support by default. The following additional lines are needed in your `local.conf` to enable it:

```
PACKAGECONFIG_pn-qemu-native = "sdl"
ASSUME_PROVIDED += "libsdl-native"
```

#### Note

The default `local.conf` contains these statements. Consequently, if you are building a headless system and using a default `local.conf` file, you will need comment these two lines out.

#### 24.5.8. core-image-basic¶

`core-image-basic` has been renamed to `core-image-full-cmdline`.

In addition to `core-image-basic` being renamed, `packagegroup-core-basic` has been renamed to `packagegroup-core-full-cmdline` to match.

#### 24.5.9. Licensing¶

The top-level `LICENSE` file has been changed to better describe the license of the various components of `OE-Core`. However, the licensing itself remains unchanged.

Normally, this change would not cause any side-effects. However, some recipes point to this file within `LIC_FILES_CHKSUM` (as `${COREBASE}/LICENSE`) and thus the accompanying checksum must be changed from `3f40d7994397109285ec7b81fdeb3b58` to `4d92cd373abda3937c2bc47fbc49d690`. A better alternative is to have `LIC_FILES_CHKSUM` point to a file describing the license that is distributed with the source that the recipe is building, if possible, rather than pointing to `${COREBASE}/LICENSE`.

#### 24.5.10. CFLAGS Options¶

The `"-fpermissive"` option has been removed from the default `CFLAGS` value. You need to take action on individual recipes that fail when building with this option. You need to either patch the recipes to fix the issues reported by the compiler, or you need to add `"-fpermissive"` to `CFLAGS` in the recipes.

#### 24.5.11. Custom Image Output Types¶

Custom image output types, as selected using `IMAGE_FSTYPES`, must declare their dependencies on other image types (if any) using a new `IMAGE_TYPEDEP` variable.

#### 24.5.12. Tasks¶

The `do_package_write` task has been removed. The task is no longer needed.

#### 24.5.13. update-alternative Provider¶

The default `update-alternatives` provider has been changed from `opkg` to `opkg-utils`. This change resolves some troublesome circular dependencies. The runtime package has also been renamed from `update-alternatives-cworth` to `update-alternatives-opkg`.

#### 24.5.14. virtclass Overrides¶

The `virtclass` overrides are now deprecated. Use the equivalent class overrides instead (e.g. `virtclass-native` becomes `class-native`.)

#### 24.5.15. Removed and Renamed Recipes¶

The following recipes have been removed:

- `packagegroup-toolset-native` - This recipe is largely unused.
- `linux-yocto-3.8` - Support for the Linux yocto 3.8 kernel has been dropped. Support for the 3.10 and 3.14 kernels have been added with the `linux-yocto-3.10` and `linux-yocto-3.14` recipes.
- `ocf-linux` - This recipe has been functionally replaced using `cryptodev-linux`.
- `genext2fs` - `genext2fs` is no longer used by the build system and is unmaintained upstream.
- `js` - This provided an ancient version of Mozilla's javascript engine that is no longer needed.
- `zaurusd` - The recipe has been moved to the `meta-handheld` layer.
- `eglibc 2.17` - Replaced by the `eglibc 2.19` recipe.
- `gcc 4.7.2` - Replaced by the now stable `gcc 4.8.2`.
- `external-sourcery-toolchain` - this recipe is now maintained in the `meta-sourcery` layer.
- `linux-libc-headers-yocto 3.4+git` - Now using version 3.10 of the `linux-libc-headers` by default.
- `meta-toolchain-gmae` - This recipe is obsolete.
- `packagegroup-core-sdk-gmae` - This recipe is obsolete.
- `packagegroup-core-standalone-gmae-sdk-target` - This recipe is obsolete.

#### 24.5.16. Removed Classes¶

The following classes have become obsolete and have been removed:

- `module_strip`
- `pkg_metainfo`
- `pkg_distribute`
- `image-empty`

#### 24.5.17. Reference Board Support Packages (BSPs)¶

The following reference BSPs changes occurred:

- The BeagleBoard (`beagleboard`) ARM reference hardware has been replaced by the BeagleBone (`beaglebone`) hardware.
- The RouterStation Pro (`routerstationpro`) MIPS reference hardware has been replaced by the EdgeRouter Lite (`edgerouter`) hardware.

The previous reference BSPs for the `beagleboard` and `routerstationpro` machines are still available in a new `meta-yocto-bsp-old` layer in the [Source Repositories](http://git.yoctoproject.org/cgit/cgit.cgi/meta-yocto-bsp-old/) at <http://git.yoctoproject.org/cgit/cgit.cgi/meta-yocto-bsp-old/>.

## 24.6. Moving to the Yocto Project 1.7 Release¶

This section provides migration information for moving to the Yocto Project 1.7 Release from the prior release.

### 24.6.1. Changes to Setting QEMU PACKAGECONFIG Options in local.conf¶

The QEMU recipe now uses a number of `PACKAGECONFIG` options to enable various optional features. The method used to set defaults for these options means that existing `local.conf` files will need to be modified to append to `PACKAGECONFIG` for `qemu-native` and `nativesdk-qemu` instead of setting it. In other words, to enable graphical output for QEMU, you should now have these lines in `local.conf`:

```
PACKAGECONFIG_append_pn-qemu-native = " sdl"
PACKAGECONFIG_append_pn-nativesdk-qemu = " sdl"
```

### 24.6.2. Minimum Git version¶

The minimum `Git` version required on the build host is now 1.7.8 because the `--list` option is now required by BitBake's `Git` fetcher. As always, if your host distribution does not provide a version of `Git` that meets this requirement, you can use the `buildtools-tarball` that does. See the "[Required Git, tar, and Python Versions](#)" section for more information.

### 24.6.3. Autotools Class Changes¶

The following `autotools` class changes occurred:

- **A separate build directory is now used by default:** The `autotools` class has been changed to use a directory for building (`B`), which is separate from the source directory (`S`). This is commonly referred to as `B != S`, or an out-of-tree build.

If the software being built is already capable of building in a directory separate from the source, you do not need to do anything. However, if the software is not capable of being built in this manner, you will need to either patch the software so that it can build separately, or you will need to change the recipe to inherit the `autotools-brokensep` class instead of the `autotools` or `autotools_stage` classes.

- **The `--foreign` option is no longer passed to `automake` when running `autoconf`:** This option tells `automake` that a particular software package does not follow the GNU standards and therefore should not be expected to distribute certain files such as `ChangeLog`, `AUTHORS`, and so forth. Because the majority of upstream software packages already tell `automake` to enable foreign mode themselves, the option is mostly superfluous. However, some recipes will need patches for this change. You can easily make the change by patching `configure.ac` so that it passes "foreign" to `AM_INIT_AUTOMAKE()`. See [this commit](#) for an example showing how to make the patch.

### 24.6.4. Binary Configuration Scripts Disabled¶

Some of the core recipes that package binary configuration scripts now disable the scripts due to the scripts previously requiring error-prone path substitution. Software that links against these libraries using these scripts should use the much more robust `pkg-config` instead. The list of recipes changed in this version (and their configuration scripts) is as follows:

```
directfb (directfb-config)
freetype (freetype-config)
gpgme (gpgme-config)
libassuan (libassuan-config)
libcroco (croco-6.0-config)
libgcrypt (libgcrypt-config)
libgpg-error (gpg-error-config)
libksba (ksba-config)
libpcap (pcap-config)
libpcre (pcre-config)
libpng (libpng-config, libpng16-config)
libsdl (sdl-config)
libusb-compat (libusb-config)
libxml2 (xml2-config)
libxslt (xslt-config)
ncurses (ncurses-config)
neon (neon-config)
npth (npth-config)
pth (pth-config)
taglib (taglib-config)
```

Additionally, support for `pkg-config` has been added to some recipes in the previous list in the rare cases where the upstream software package does not already provide it.

### 24.6.5. `eglibc 2.19` Replaced with `glibc 2.20`¶

Because `eglibc` and `glibc` were already fairly close, this replacement should not require any significant changes to other software that links to `eglibc`. However, there were a number of minor changes in `glibc 2.20` upstream that could require patching some software (e.g. the removal of the `_BSD_SOURCE` feature test macro).

`glibc 2.20` requires version 2.6.32 or greater of the Linux kernel. Thus, older kernels will no longer be usable in conjunction with it.

For full details on the changes in `glibc 2.20`, see the upstream release notes [here](#).

### 24.6.6. Kernel Module Autoloading¶

The `module_autoload_*` variable is now deprecated and a new `KERNEL_MODULE_AUTOLOAD` variable should be used instead. Also, `module_conf_*` must now be used in conjunction with a new `KERNEL_MODULE_PROBECONF` variable.

The new variables no longer require you to specify the module name as part of the variable name. This change not only simplifies usage but also allows the values of these variables to be appropriately incorporated into task signatures and thus trigger the appropriate tasks

to re-execute when changed. You should replace any references to `module_autoload *` with `KERNEL_MODULE_AUTOLOAD`, and add any modules for which `module_conf_*` is specified to `KERNEL_MODULE_PROBECONF`.

### 24.6.7. QA Check Changes¶

The following changes have occurred to the QA check process:

- Additional QA checks `file-rdeps` and `build-deps` have been added in order to verify that file dependencies are satisfied (e.g. package contains a script requiring `/bin/bash`) and build-time dependencies are declared, respectively. For more information, please see the "QA Error and Warning Messages" chapter.
- Package QA checks are now performed during a new `do_package_qa` task rather than being part of the `do_package` task. This allows more parallel execution. This change is unlikely to be an issue except for highly customized recipes that disable packaging tasks themselves by marking them as `noexec`. For those packages, you will need to disable the `do_package_qa` task as well.
- Files being overwritten during the `do_populate_sysroot` task now trigger an error instead of a warning. Recipes should not be overwriting files written to the sysroot by other recipes. If you have these types of recipes, you need to alter them so that they do not overwrite these files.

You might now receive this error after changes in configuration or metadata resulting in orphaned files being left in the sysroot. If you do receive this error, the way to resolve the issue is to delete your `TMPDIR` or to move it out of the way and then re-start the build. Anything that has been fully built up to that point and does not need rebuilding will be restored from the shared state cache and the rest of the build will be able to proceed as normal.

### 24.6.8. Removed Recipes¶

The following recipes have been removed:

- `x-load`: This recipe has been superseded by U-boot SPL for all Cortex-based TI SoCs. For legacy boards, the `meta-ti` layer, which contains a maintained recipe, should be used instead.
- `ubootchart`: This recipe is obsolete. A `bootchart2` recipe has been added to functionally replace it.
- `linux-yocto 3.4`: Support for the linux-yocto 3.4 kernel has been dropped. Support for the 3.10 and 3.14 kernels remains, while support for version 3.17 has been added.
- `eglibc` has been removed in favor of `glibc`. See the "[eglibc 2.19 Replaced with glibc 2.20](#)" section for more information.

### 24.6.9. Miscellaneous Changes¶

The following miscellaneous change occurred:

- The build history feature now writes `build-id.txt` instead of `build-id`. Additionally, `build-id.txt` now contains the full build header as printed by BitBake upon starting the build. You should manually remove old "build-id" files from your existing build history repositories to avoid confusion. For information on the build history feature, see the "[Maintaining Build Output Quality](#)" section in the Yocto Project Development Tasks Manual.

## 24.7. Moving to the Yocto Project 1.8 Release¶

This section provides migration information for moving to the Yocto Project 1.8 Release from the prior release.

### 24.7.1. Removed Recipes¶

The following recipes have been removed:

- `owl-video`: Functionality replaced by `gst-player`.
- `gaku`: Functionality replaced by `gst-player`.
- `gnome-desktop`: This recipe is now available in `meta-gnome` and is no longer needed.
- `gsettings-desktop-schemas`: This recipe is now available in `meta-gnome` and is no longer needed.
- `python-argparse`: The `argparse` module is already provided in the default Python distribution in a package named `python-argparse`. Consequently, the separate `python-argparse` recipe is no longer needed.
- `telepathy-python`, `libtelepathy`, `telepathy-glib`, `telepathy-idle`, `telepathy-mission-control`: All these recipes have moved to `meta-oe` and are consequently no longer needed by any recipes in `OpenEmbedded-Core`.
- `linux-yocto_3.10` and `linux-yocto_3.17`: Support for the linux-yocto 3.10 and 3.17 kernels has been dropped. Support for the 3.14 kernel remains, while support for 3.19 kernel has been added.
- `poky-feed-config-opkg`: This recipe has become obsolete and is no longer needed. Use `distro-feed-config` from `meta-oe` instead.
- `libav 0.8.x`: `libav 9.x` is now used.
- `sed-native`: No longer needed. A working version of `sed` is expected to be provided by the host distribution.

### 24.7.2. BlueZ 4.x / 5.x Selection¶

Proper built-in support for selecting BlueZ 5.x in preference to the default of 4.x now exists. To use BlueZ 5.x, simply add "bluez5" to your `DISTRO_FEATURES` value. If you had previously added append files (`*.bbappend`) to make this selection, you can now

remove them.

Additionally, a `bluetooth` class has been added to make selection of the appropriate bluetooth support within a recipe a little easier. If you wish to make use of this class in a recipe, add something such as the following:

```
inherit bluetooth
PACKAGECONFIG ??= "${@bb.utils.contains('DISTRO_FEATURES', 'bluetooth', '${BLUEZ}', '', d)}
PACKAGECONFIG[bluez4] = "--enable-bluetooth,--disable-bluetooth,bluez4"
PACKAGECONFIG[bluez5] = "--enable-bluez5,--disable-bluez5,bluez5"
```

### 24.7.3. Kernel Build Changes¶

The kernel build process was changed to place the source in a common shared work area and to place build artifacts separately in the source code tree. In theory, migration paths have been provided for most common usages in kernel recipes but this might not work in all cases. In particular, users need to ensure that `_${S}` (source files) and `_${B}` (build artifacts) are used correctly in functions such as `do_configure` and `do_install`. For kernel recipes that do not inherit from `kernel-yocto` or include `linux-yocto.inc`, you might wish to refer to the `linux.inc` file in the `meta-oe` layer for the kinds of changes you need to make. For reference, here is the [commit](#) where the `linux.inc` file in `meta-oe` was updated.

Recipes that rely on the kernel source code and do not inherit the module classes might need to add explicit dependencies on the `do_shared_workdir` kernel task, for example:

```
do_configure[depends] += "virtual/kernel:do_shared_workdir"
```

### 24.7.4. SSL 3.0 is Now Disabled in OpenSSL¶

SSL 3.0 is now disabled when building OpenSSL. Disabling SSL 3.0 avoids any lingering instances of the POODLE vulnerability. If you feel you must re-enable SSL 3.0, then you can add an append file (`*.bbappend`) for the `openssl` recipe to remove `"-no-ssl3"` from `EXTRA_OECONF`.

### 24.7.5. Default Sysroot Poisoning¶

`gcc`'s default sysroot and include directories are now "poisoned". In other words, the sysroot and include directories are being redirected to a non-existent location in order to catch when host directories are being used due to the correct options not being passed. This poisoning applies both to the cross-compiler used within the build and to the cross-compiler produced in the SDK.

If this change causes something in the build to fail, it almost certainly means the various compiler flags and commands are not being passed correctly to the underlying piece of software. In such cases, you need to take corrective steps.

### 24.7.6. Rebuild Improvements¶

Changes have been made to the `base`, `autotools`, and `cmake` classes to clean out generated files when the `do_configure` task needs to be re-executed.

One of the improvements is to attempt to run "make clean" during the `do_configure` task if a `Makefile` exists. Some software packages do not provide a working clean target within their make files. If you have such recipes, you need to set `CLEANBROKEN` to "1" within the recipe, for example:

```
CLEANBROKEN = "1"
```

### 24.7.7. QA Check and Validation Changes¶

The following QA Check and Validation Changes have occurred:

- Usage of `PRINC` previously triggered a warning. It now triggers an error. You should remove any remaining usage of `PRINC` in any recipe or append file.
- An additional QA check has been added to detect usage of `_${D}` in `FILES` values where `_${D}` values should not be used at all. The same check ensures that `_${D}` is used in `pkg_preinst/pkg_postinst/pkg_prerm/pkg_postrm` functions instead of `_${D}`.
- `_${S}` now needs to be set to a valid value within a recipe. If `_${S}` is not set in the recipe, the directory is not automatically created. If `_${S}` does not point to a directory that exists at the time the `do_unpack` task finishes, a warning will be shown.
- `LICENSE` is now validated for correct formatting of multiple licenses. If the format is invalid (e.g. multiple licenses are specified with no operators to specify how the multiple licenses interact), then a warning will be shown.

### 24.7.8. Miscellaneous Changes¶

The following miscellaneous changes have occurred:

- The `send-error-report` script now expects a `"-s"` option to be specified before the server address. This assumes a server address is being specified.
- The `oe-pkgdata-util` script now expects a `"-p"` option to be specified before the `pkgdata` directory, which is now optional. If the `pkgdata` directory is not specified, the script will run BitBake to query `PKGDATA_DIR` from the build environment.

## 24.8. Moving to the Yocto Project 2.0 Release¶

This section provides migration information for moving to the Yocto Project 2.0 Release from the prior release.

### 24.8.1. GCC 5¶

The default compiler is now GCC 5.2. This change has required fixes for compilation errors in a number of other recipes.

One important example is a fix for when the Linux kernel freezes at boot time on ARM when built with GCC 5. If you are using your own kernel recipe or source tree and building for ARM, you will likely need to apply this [patch](#). The standard `linux-yocto` kernel source tree already has a workaround for the same issue.

For further details, see <https://gcc.gnu.org/gcc-5/changes.html> and the porting guide at [https://gcc.gnu.org/gcc-5/porting\\_to.html](https://gcc.gnu.org/gcc-5/porting_to.html).

Alternatively, you can switch back to GCC 4.9 or 4.8 by setting `GCCVERSION` in your configuration, as follows:

```
GCCVERSION = "4.9"
```

### 24.8.2. Gstreamer 0.10 Removed¶

Gstreamer 0.10 has been removed in favor of Gstreamer 1.x. As part of the change, recipes for Gstreamer 0.10 and related software are now located in `meta-multimedia`. This change results in Qt4 having Phonon and Gstreamer support in QtWebkit disabled by default.

### 24.8.3. Removed Recipes¶

The following recipes have been moved or removed:

- `bluez4`: The recipe is obsolete and has been moved due to `bluez5` becoming fully integrated. The `bluez4` recipe now resides in `meta-oe`.
- `gamin`: The recipe is obsolete and has been removed.
- `gnome-icon-theme`: The recipe's functionality has been replaced by `adwaita-icon-theme`.
- Gstreamer 0.10 Recipes: Recipes for Gstreamer 0.10 have been removed in favor of the recipes for Gstreamer 1.x.
- `insserv`: The recipe is obsolete and has been removed.
- `libunique`: The recipe is no longer used and has been moved to `meta-oe`.
- `midori`: The recipe's functionality has been replaced by `epiphany`.
- `python-gst`: The recipe is obsolete and has been removed since it only contains bindings for Gstreamer 0.10.
- `qt-mobility`: The recipe is obsolete and has been removed since it requires Gstreamer 0.10, which has been replaced.
- `subversion`: All 1.6.x versions of this recipe have been removed.
- `webkit-gtk`: The older 1.8.3 version of this recipe has been removed in favor of `webkitgtk`.

### 24.8.4. BitBake datastore improvements¶

The method by which BitBake's datastore handles overrides has changed. Overrides are now applied dynamically and `bb.data.update_data()` is now a no-op. Thus, `bb.data.update_data()` is no longer required in order to apply the correct overrides. In practice, this change is unlikely to require any changes to Metadata. However, these minor changes in behavior exist:

- All potential overrides are now visible in the variable history as seen when you run the following:
 

```
$ bitbake -e
```
- `d.delVar('VARNAME')` and `d.setVar('VARNAME', None)` result in the variable and all of its overrides being cleared out. Before the change, only the non-overridden values were cleared.

### 24.8.5. Shell Message Function Changes¶

The shell versions of the BitBake message functions (i.e. `bbdebug`, `bbnote`, `bbwarn`, `bbplain`, `bberror`, and `bbfatal`) are now connected through to their BitBake equivalents `bb.debug()`, `bb.note()`, `bb.warn()`, `bb.plain()`, `bb.error()`, and `bb.fatal()`, respectively. Thus, those message functions that you would expect to be printed by the BitBake UI are now actually printed. In practice, this change means two things:

- If you now see messages on the console that you did not previously see as a result of this change, you might need to clean up the calls to `bbwarn`, `bberror`, and so forth. Or, you might want to simply remove the calls.
- The `bbfatal` message function now suppresses the full error log in the UI, which means any calls to `bbfatal` where you still wish to see the full error log should be replaced by `die` or `bbfatal_log`.

### 24.8.6. Extra Development/Debug Package Cleanup¶

The following recipes have had extra `dev/dbg` packages removed:

- `acl`
- `apmd`
- `aspell`
- `attr`
- `augeas`

- `bzip2`
- `cogl`
- `curl`
- `elfutils`
- `gcc-target`
- `libgcc`
- `libtool`
- `libxmu`
- `opkg`
- `pciutils`
- `rpm`
- `sysfsutils`
- `tiff`
- `xz`

All of the above recipes now conform to the standard packaging scheme where a single `-dev`, `-dbg`, and `-staticdev` package exists per recipe.

#### 24.8.7. Recipe Maintenance Tracking Data Moved to OE-Core¶

Maintenance tracking data for recipes that was previously part of `meta-yocto` has been moved to [OE-Core](#). The change includes `package_regex.inc` and `distro_alias.inc`, which are typically enabled when using the `distrodata` class. Additionally, the contents of `upstream_tracking.inc` has now been split out to the relevant recipes.

#### 24.8.8. Automatic Stale Sysroot File Cleanup¶

Stale files from recipes that no longer exist in the current configuration are now automatically removed from sysroot as well as removed from any other place managed by shared state. This automatic cleanup means that the build system now properly handles situations such as renaming the build system side of recipes, removal of layers from `bblayers.conf`, and [DISTRO\\_FEATURES](#) changes.

Additionally, work directories for old versions of recipes are now pruned. If you wish to disable pruning old work directories, you can set the following variable in your configuration:

```
SSTATE_PRUNE_OBSOLETEWORKDIR = "0"
```

#### 24.8.9. linux-yocto Kernel Metadata Repository Now Split from Source¶

The `linux-yocto` tree has up to now been a combined set of kernel changes and configuration (meta) data carried in a single tree. While this format is effective at keeping kernel configuration and source modifications synchronized, it is not always obvious to developers how to manipulate the Metadata as compared to the source.

Metadata processing has now been removed from the `kernel-yocto` class and the external Metadata repository `yocto-kernel-cache`, which has always been used to seed the `linux-yocto` "meta" branch. This separate `linux-yocto` cache repository is now the primary location for this data. Due to this change, `linux-yocto` is no longer able to process combined trees. Thus, if you need to have your own combined kernel repository, you must do the split there as well and update your recipes accordingly. See the `meta/recipes-kernel/linux/linux-yocto_4.1.bb` recipe for an example.

#### 24.8.10. Additional QA checks¶

The following QA checks have been added:

- Added a "host-user-contaminated" check for ownership issues for packaged files outside of `/home`. The check looks for files that are incorrectly owned by the user that ran BitBake instead of owned by a valid user in the target system.
- Added an "invalid-chars" check for invalid (non-UTF8) characters in recipe metadata variable values (i.e. [DESCRIPTION](#), [SUMMARY](#), [LICENSE](#), and [SECTION](#)). Some package managers do not support these characters.
- Added an "invalid-packageconfig" check for any options specified in [PACKAGECONFIG](#) that do not match any [PACKAGECONFIG](#) option defined for the recipe.

#### 24.8.11. Miscellaneous Changes¶

These additional changes exist:

- `gtk-update-icon-cache` has been renamed to `gtk-icon-utils`.
- The tools-profile [IMAGE\\_FEATURES](#) item as well as its corresponding packagegroup and `packagegroup-core-tools-profile` no longer bring in `oprofile`. Bringing in `oprofile` was originally added to aid compilation on resource-constrained targets. However, this aid has not been widely used and is not likely to be used going forward due to the more powerful target platforms and the existence of better cross-compilation tools.
- The [IMAGE\\_FSTYPES](#) variable's default value now specifies `ext4` instead of `ext3`.
- All support for the `PRINC` variable has been removed.



- The `packagegroup-core-full-cmdline` packagegroup no longer brings in `lighttpd` due to the fact that bringing in `lighttpd` is not really in line with the packagegroup's purpose, which is to add full versions of command-line tools that by default are provided by `busybox`.

## 24.9. Moving to the Yocto Project 2.1 Release¶

This section provides migration information for moving to the Yocto Project 2.1 Release from the prior release.

### 24.9.1. Variable Expansion in Python Functions¶

Variable expressions, such as `${VARNAME}` no longer expand automatically within Python functions. Suppressing expansion was done to allow Python functions to construct shell scripts or other code for situations in which you do not want such expressions expanded. For any existing code that relies on these expansions, you need to change the expansions to expand the value of individual variables through `d.getVar()`. To alternatively expand more complex expressions, use `d.expand()`.

### 24.9.2. Overrides Must Now be Lower-Case¶

The convention for overrides has always been for them to be lower-case characters. This practice is now a requirement as BitBake's datastore now assumes lower-case characters in order to give a slight performance boost during parsing. In practical terms, this requirement means that anything that ends up in `OVERRIDES` must now appear in lower-case characters (e.g. values for `MACHINE`, `TARGET_ARCH`, `DISTRO`, and also recipe names if `_pn-recipe` overrides are to be effective).

### 24.9.3. Expand Parameter to `getVar()` and `getVarFlag()` is Now Mandatory¶

The `expand` parameter to `getVar()` and `getVarFlag()` previously defaulted to `False` if not specified. Now, however, no default exists so one must be specified. You must change any `getVar()` calls that do not specify the final `expand` parameter to calls that do specify the parameter. You can run the following `sed` command at the base of a layer to make this change:

```
sed -e 's:(\.\.getVar([^\,()]*)):\\1, False):g' -i `grep -ril getVar *`
sed -e 's:(\.\.getVarFlag([^\,()]*, [^\,()]*)):\\1, False):g' -i `grep -ril getVarFlag *`
```

#### Note

The reason for this change is that it prepares the way for changing the default to `True` in a future Yocto Project release. This future change is a much more sensible default than `False`. However, the change needs to be made gradually as a sudden change of the default would potentially cause side-effects that would be difficult to detect.

### 24.9.4. Makefile Environment Changes¶

`EXTRA_OEMAKE` now defaults to `""` instead of `"-e MAKEFLAGS="`. Setting `EXTRA_OEMAKE` to `"-e MAKEFLAGS="` by default was a historical accident that has required many classes (e.g. `autotools`, `module`) and recipes to override this default in order to work with sensible build systems. When upgrading to the release, you must edit any recipe that relies upon this old default by either setting `EXTRA_OEMAKE` back to `"-e MAKEFLAGS="` or by explicitly setting any required variable value overrides using `EXTRA_OEMAKE`, which is typically only needed when a Makefile sets a default value for a variable that is inappropriate for cross-compilation using the `"="` operator rather than the `"?="` operator.

### 24.9.5. `libexecdir` Reverted to `${prefix}/libexec`¶

The use of `${libdir}/${BPN}` as `libexecdir` is different as compared to all other mainstream distributions, which either uses `${prefix}/libexec` or `${libdir}`. The use is also contrary to the GNU Coding Standards (i.e. [https://www.gnu.org/prep/standards/html\\_node/Directory-Variables.html](https://www.gnu.org/prep/standards/html_node/Directory-Variables.html)) that suggest `${prefix}/libexec` and also notes that any package-specific nesting should be done by the package itself. Finally, having `libexecdir` change between recipes makes it very difficult for different recipes to invoke binaries that have been installed into `libexecdir`. The Filesystem Hierarchy Standard (i.e. [http://refspecs.linuxfoundation.org/FHS\\_3.0/fhs/ch04s07.html](http://refspecs.linuxfoundation.org/FHS_3.0/fhs/ch04s07.html)) now recognizes the use of `${prefix}/libexec/`, giving distributions the choice between `${prefix}/lib` or `${prefix}/libexec` without breaking FHS.

### 24.9.6. `ac_cv_sizeof_off_t` is No Longer Cached in Site Files¶

For recipes inheriting the `autotools` class, `ac_cv_sizeof_off_t` is no longer cached in the site files for `autoconf`. The reason for this change is because the `ac_cv_sizeof_off_t` value is not necessarily static per architecture as was previously assumed. Rather, the value changes based on whether large file support is enabled. For most software that uses `autoconf`, this change should not be a problem. However, if you have a recipe that bypasses the standard `do_configure` task from the `autotools` class and the software the recipe is building uses a very old version of `autoconf`, the recipe might be incapable of determining the correct size of `off_t` during `do_configure`.

The best course of action is to patch the software as necessary to allow the default implementation from the `autotools` class to work such that `autoreconf` succeeds and produces a working configure script, and to remove the overridden `do_configure` task such that the default implementation does get used.

### 24.9.7. Image Generation is Now Split Out from Filesystem Generation¶

Previously, for image recipes the `do_rootfs` task assembled the filesystem and then from that filesystem generated images. With this Yocto Project release, image generation is split into separate `do_image_*` tasks for clarity both in operation and in the code.

For most cases, this change does not present any problems. However, if you have made customizations that directly modify the `do_rootfs` task or that mention `do_rootfs`, you might need to update those changes. In particular, if you had added any



tasks after `do_rootfs`, you should make edits so that those tasks are after the `do_image_complete` task rather than after `do_rootfs` so that the your added tasks run at the correct time.

A minor part of this restructuring is that the post-processing definitions and functions have been moved from the `image` class to the `rootfs-postcommands` class. Functionally, however, they remain unchanged.

### 24.9.8. Removed Recipes¶

The following recipes have been removed in the 2.1 release:

- `gcc` version 4.8: Versions 4.9 and 5.3 remain.
- `qt4`: All support for Qt 4.x has been moved out to a separate `meta-qt4` layer because Qt 4 is no longer supported upstream.
- `x11vnc`: Moved to the `meta-oe` layer.
- `linux-yocto-3.14`: No longer supported.
- `linux-yocto-3.19`: No longer supported.
- `libjpeg`: Replaced by the `libjpeg-turbo` recipe.
- `pth`: Became obsolete.
- `liboil`: Recipe is no longer needed and has been moved to the `meta-multimedia` layer.
- `gtk-theme-torturer`: Recipe is no longer needed and has been moved to the `meta-gnome` layer.
- `gnome-mime-data`: Recipe is no longer needed and has been moved to the `meta-gnome` layer.
- `udev`: Replaced by the `eudev` recipe for compatibility when using `sysvinit` with newer kernels.
- `python-pygtk`: Recipe became obsolete.
- `adt-installer`: Recipe became obsolete. See the "[ADT Removed](#)" section for more information.

### 24.9.9. Class Changes¶

The following classes have changed:

- `autotools_stage`: Removed because the `autotools` class now provides its functionality. Recipes that inherited from `autotools_stage` should now inherit from `autotools` instead.
- `boot-directdisk`: Merged into the `image-vm` class. The `boot-directdisk` class was rarely directly used. Consequently, this change should not cause any issues.
- `bootimg`: Merged into the `image-live` class. The `bootimg` class was rarely directly used. Consequently, this change should not cause any issues.
- `packageinfo`: Removed due to its limited use by the Hob UI, which has itself been removed.

### 24.9.10. Build System User Interface Changes¶

The following changes have been made to the build system user interface:

- **Hob GTK+-based UI**: Removed because it is unmaintained and based on the outdated GTK+ 2 library. The Toaster web-based UI is much more capable and is actively maintained. See the "[Using the Toaster Web Interface](#)" section in the Toaster User Manual for more information on this interface.
- **"puccho" BitBake UI**: Removed because is unmaintained and no longer useful.

### 24.9.11. ADT Removed¶

The Application Development Toolkit (ADT) has been removed because its functionality almost completely overlapped with the [standard SDK](#) and the [extensible SDK](#). For information on these SDKs and how to build and use them, see the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

## Note

The Yocto Project Eclipse IDE Plug-in is still supported and is not affected by this change.

### 24.9.12. Poky Reference Distribution Changes¶

The following changes have been made for the Poky distribution:

- The `meta-yocto` layer has been renamed to `meta-poky` to better match its purpose, which is to provide the Poky reference distribution. The `meta-yocto-bsp` layer retains its original name since it provides reference machines for the Yocto Project and it is otherwise unrelated to Poky. References to `meta-yocto` in your `conf/bblayers.conf` should automatically be updated, so you should not need to change anything unless you are relying on this naming elsewhere.
- The `uninative` class is now enabled by default in Poky. This class attempts to isolate the build system from the host distribution's C library and makes re-use of native shared state artifacts across different host distributions practical. With this class enabled, a tarball containing a pre-built C library is downloaded at the start of the build.

The `uninative` class is enabled through the `meta/conf/distro/include/yocto-uninative.inc` file, which for those not using the Poky distribution, can include to easily enable the same functionality.

Alternatively, if you wish to build your own `uninative` tarball, you can do so by building the `uninative-tarball` recipe, making it available to your build machines (e.g. over HTTP/HTTPS) and setting a similar configuration as the one set by `yocto-uninative.inc`.

- Static library generation, for most cases, is now disabled by default in the Poky distribution. Disabling this generation saves some build time as well as the size used for build output artifacts.

Disabling this library generation is accomplished through a `meta/conf/distro/include/no-static-libs.inc`, which for those not using the Poky distribution can easily include to enable the same functionality.

Any recipe that needs to opt-out of having the `--disable-static` option specified on the configure command line either because it is not a supported option for the configure script or because static libraries are needed should set the following variable:

```
DISABLE_STATIC = ""
```

- The separate `poky-tiny` distribution now uses the musl C library instead of a heavily pared down `glibc`. Using musl results in a smaller distribution and facilitates much greater maintainability because musl is designed to have a small footprint.

If you have used `poky-tiny` and have customized the `glibc` configuration you will need to redo those customizations with musl when upgrading to the new release.

### 24.9.13. Packaging Changes¶

The following changes have been made to packaging:

- The `runuser` and `mountpoint` binaries, which were previously in the main `util-linux` package, have been split out into the `util-linux-runuser` and `util-linux-mountpoint` packages, respectively.
- The `python-elementtree` package has been merged into the `python-xml` package.

### 24.9.14. Tuning File Changes¶

The following changes have been made to the tuning files:

- The "no-thumb-interwork" tuning feature has been dropped from the ARM tune include files. Because interworking is required for ARM EABI, attempting to disable it through a tuning feature no longer makes sense.

#### Note

Support for ARM OABI was deprecated in gcc 4.7.

- The `tune-cortexm*.inc` and `tune-cortexr4.inc` files have been removed because they are poorly tested. Until the OpenEmbedded build system officially gains support for CPUs without an MMU, these tuning files would probably be better maintained in a separate layer if needed.

### 24.9.15. Supporting GObject Introspection¶

This release supports generation of GLib Introspective Repository (GIR) files through GObject introspection, which is the standard mechanism for accessing GObject-based software from runtime environments. You can enable, disable, and test the generation of this data. See the "[Enabling GObject Introspection Support](#)" section in the Yocto Project Development Tasks Manual for more information.

### 24.9.16. Miscellaneous Changes¶

These additional changes exist:

- The minimum Git version has been increased to 1.8.3.1. If your host distribution does not provide a sufficiently recent version, you can install the `buildtools`, which will provide it. See the "[Required Git, tar, and Python Versions](#)" section for more information on the `buildtools` tarball.
- The buggy and incomplete support for the RPM version 4 package manager has been removed. The well-tested and maintained support for RPM version 5 remains.
- Previously, the following list of packages were removed if package-management was not in `IMAGE_FEATURES`, regardless of any dependencies:

```
update-rc.d
base-passwd
shadow
update-alternatives
run-postinsts
```

With the Yocto Project 2.1 release, these packages are only removed if "read-only-rootfs" is in `IMAGE_FEATURES`, since they might still be needed for a read-write image even in the absence of a package manager (e.g. if users need to be added, modified, or removed at runtime).

- The `devtool modify` command now defaults to extracting the source since that is most commonly expected. The `-x` or `--extract` options are now no-ops. If you wish to provide your own existing source tree, you will now need to specify either the `-n` or `--no-extract` options when running `devtool modify`.
- If the `formfactor` for a machine is either not supplied or does not specify whether a keyboard is attached, then the default is to assume a keyboard is attached rather than assume no keyboard. This change primarily affects the Sato UI.
- The `.debug` directory packaging is now automatic. If your recipe builds software that installs binaries into directories other than the standard ones, you no longer need to take care of setting `FILES_${PN}-dbg` to pick up the resulting `.debug` directories as these directories are automatically found and added.

- Inaccurate disk and CPU percentage data has been dropped from `buildstats` output. This data has been replaced with `getrusage()` data and corrected IO statistics. You will probably need to update any custom code that reads the `buildstats` data.
- The `meta/conf/distro/include/package_regex.inc` is now deprecated. The contents of this file have been moved to individual recipes.

### Tip

Because this file will likely be removed in a future Yocto Project release, it is suggested that you remove any references to the file that might be in your configuration.

- The `v86d/uvesafb` has been removed from the `genericx86` and `genericx86-64` reference machines, which are provided by the `meta-yocto-bsp` layer. Most modern x86 boards do not rely on this file and it only adds kernel error messages during startup. If you do still need to support `uvesafb`, you can simply add `v86d` to your image.
- Build sysroot paths are now removed from debug symbol files. Removing these paths means that remote GDB using an unstripped build system sysroot will no longer work (although this was never documented to work). The supported method to accomplish something similar is to set `IMAGE_GEN_DEBUGFS` to "1", which will generate a companion debug image containing unstripped binaries and associated debug sources alongside the image.

## 24.10. Moving to the Yocto Project 2.2 Release¶

This section provides migration information for moving to the Yocto Project 2.2 Release from the prior release.

### 24.10.1. Minimum Kernel Version¶

The minimum kernel version for the target system and for SDK is now 3.2.0, due to the upgrade to `glibc 2.24`. Specifically, for AArch64-based targets the version is 3.14. For Nios II-based targets, the minimum kernel version is 3.19.

### Note

For x86 and x86\_64, you can reset `OLDEST_KERNEL` to anything down to 2.6.32 if desired.

### 24.10.2. Staging Directories in Sysroot Has Been Simplified¶

The way directories are staged in sysroot has been simplified and introduces the new `SYSROOT_DIRS`, `SYSROOT_DIRS_NATIVE`, and `SYSROOT_DIRS_BLACKLIST`. See the [v2 patch series on the OE-Core Mailing List](#) for additional information.

### 24.10.3. Removal of Old Images and Other Files in `tmp/deploy` Now Enabled¶

Removal of old images and other files in `tmp/deploy/` is now enabled by default due to a new staging method used for those files. As a result of this change, the `RM_OLD_IMAGE` variable is now redundant.

### 24.10.4. Python Changes¶

The following changes for Python occurred:

#### 24.10.4.1. BitBake Now Requires Python 3.4+¶

BitBake requires Python 3.4 or greater.

#### 24.10.4.2. UTF-8 Locale Required on Build Host¶

A UTF-8 locale is required on the build host due to Python 3. Since C.UTF-8 is not a standard, the default is `en_US.UTF-8`.

#### 24.10.4.3. Metadata Must Now Use Python 3 Syntax¶

The metadata is now required to use Python 3 syntax. For help preparing metadata, see any of the many Python 3 porting guides available. Alternatively, you can reference the conversion commits for Bitbake and you can use [OE-Core](#) as a guide for changes. Following are particular areas of interest:

- \* subprocess command-line pipes needing locale decoding
- \* the syntax for octal values changed
- \* the `iter*()` functions changed name
- \* iterators now return views, not lists
- \* changed names for Python modules

#### 24.10.4.4. Target Python Recipes Switched to Python 3¶

Most target Python recipes have now been switched to Python 3. Unfortunately, systems using RPM as a package manager and providing online package-manager support through SMART still require Python 2.

### Note

Python 2 and recipes that use it can still be built for the target as with previous versions.

#### 24.10.4.5. buildtools-tarball Includes Python 3¶

`buildtools-tarball` now includes Python 3.

#### 24.10.5. uClibc Replaced by musl¶

uClibc has been removed in favor of musl. Musl has matured, is better maintained, and is compatible with a wider range of applications as compared to uClibc.

#### 24.10.6. \${B} No Longer Default Working Directory for Tasks¶

`${B}` is no longer the default working directory for tasks. Consequently, any custom tasks you define now need to either have the `[dirs]` flag set, or the task needs to change into the appropriate working directory manually (e.g using `cd` for a shell task).

### Note

The preferred method is to use the `[dirs]` flag.

#### 24.10.7. runqemu Ported to Python¶

`runqemu` has been ported to Python and has changed behavior in some cases. Previous usage patterns continue to be supported.

The new `runqemu` is a Python script. Machine knowledge is no longer hardcoded into `runqemu`. You can choose to use the `qemuboot` configuration file to define the BSP's own arguments and to make it bootable with `runqemu`. If you use a configuration file, use the following form:

```
image-name-machine.qemuboot.conf
```

The configuration file enables fine-grained tuning of options passed to QEMU without the `runqemu` script hard-coding any knowledge about different machines. Using a configuration file is particularly convenient when trying to use QEMU with machines other than the `qemu*` machines in [QE-Core](#). The `qemuboot.conf` file is generated by the `qemuboot` class when the root filesystem is being build (i.e. build rootfs). QEMU boot arguments can be set in BSP's configuration file and the `qemuboot` class will save them to `qemuboot.conf`.

If you want to use `runqemu` without a configuration file, use the following command form:

```
$ runqemu machine rootfs kernel [options]
```

Supported *machines* are as follows:

```
qemuarm
qemuarm64
qemux86
qemux86-64
qemuppc
qemumips
qemumips64
qemumipse1
qemumips64e1
```

Consider the following example, which uses the `qemux86-64` machine, provides a root filesystem, provides an image, and uses the `nographic` option:

```
$ runqemu qemux86-64 tmp/deploy/images/qemux86-64/core-image-minimal-qemux86-64.ext4 tmp/deploy/images/qemux86-64/bzImage nogr
```

Following is a list of variables that can be set in configuration files such as `bsp.conf` to enable the BSP to be booted by `runqemu`:

### Note

"QB" means "QEMU Boot".

```
QB_SYSTEM_NAME: QEMU name (e.g. "qemu-system-i386")
QB_OPT_APPEND: Options to append to QEMU (e.g. "-show-cursor")
QB_DEFAULT_KERNEL: Default kernel to boot (e.g. "bzImage")
QB_DEFAULT_FSTYPE: Default FSTYPE to boot (e.g. "ext4")
QB_MEM: Memory (e.g. "-m 512")
QB_MACHINE: QEMU machine (e.g. "-machine virt")
QB_CPU: QEMU cpu (e.g. "-cpu qemu32")
QB_CPU_KVM: Similar to QB_CPU except used for kvm support (e.g. "-cpu kvm64")
QB_KERNEL_CMDLINE_APPEND: Options to append to the kernel's -append
 option (e.g. "console=ttyS0 console=tty")
QB_DTB: QEMU dtb name
QB_AUDIO_DRV: QEMU audio driver (e.g. "alsa", set it when support audio)
QB_AUDIO_OPT: QEMU audio option (e.g. "-soundhw ac97,es1370"), which is used
 when QB_AUDIO_DRV is set.
QB_KERNEL_ROOT: Kernel's root (e.g. /dev/vda)
QB_TAP_OPT: Network option for 'tap' mode (e.g.
```

```

-netdev tap,id=net0,ifname=@TAP@,script=no,downscript=no -device virtio-net-device,netdev=net0").
runqemu will replace "@TAP@" with the one that is used, such as tap0, tap1 ...
QB_SLIRP_OPT: Network option for SLIRP mode (e.g. "-netdev user,id=net0 -device virtio-net-device,netdev=net0")
QB_ROOTFS_OPT: Used as rootfs (e.g.
 "-drive id=disk0,file=@ROOTFS@,if=none,format=raw -device virtio-blk-device,drive=disk0").
runqemu will replace "@ROOTFS@" with the one which is used, such as
 core-image-minimal-qemuarm64.ext4.
QB_SERIAL_OPT: Serial port (e.g. "-serial mon:stdio")
QB_TCPSERIAL_OPT: tcp serial port option (e.g.
 "-device virtio-serial-device -chardev socket,id=virtcon,port=@PORT@,host=127.0.0.1 -device virtio-serial-device,netdev=net0").
runqemu will replace "@PORT@" with the port number which is used.

```

To use `runqemu`, set `IMAGE_CLASSES` as follows and run `runqemu`:

## Note

For command-line syntax, use `runqemu help`.

```
IMAGE_CLASSES += "qemuboot"
```

### 24.10.8. Default Linker Hash Style Changed¶

The default linker hash style for `gcc-cross` is now "sysv" in order to catch recipes that are building software without using the OpenEmbedded `LDFLAGS`. This change could result in seeing some "No GNU\_HASH in the elf binary" QA issues when building such recipes. You need to fix these recipes so that they use the expected `LDFLAGS`. Depending on how the software is built, the build system used by the software (e.g. a Makefile) might need to be patched. However, sometimes making this fix is as simple as adding the following to the recipe:

```
TARGET_CC_ARCH += "${LDFLAGS}"
```

### 24.10.9. KERNEL\_IMAGE\_BASE\_NAME no Longer Uses KERNEL\_IMAGETYPE¶

The `KERNEL_IMAGE_BASE_NAME` variable no longer uses the `KERNEL_IMAGETYPE` variable to create the image's base name. Because the OpenEmbedded build system can now build multiple kernel image types, this part of the kernel image base name has been removed leaving only the following:

```
KERNEL_IMAGE_BASE_NAME ?= "${PKG}-${PKGVERSION}-${PKGNAME}-${MACHINE}-${DATETIME}"
```

If you have recipes or classes that use `KERNEL_IMAGE_BASE_NAME` directly, you might need to update the references to ensure they continue to work.

### 24.10.10. BitBake Changes¶

The following changes took place for BitBake:

- The "goggle" UI and standalone image-writer tool have been removed as they both require GTK+ 2.0 and were not being maintained.
- The Perforce fetcher now supports `SRCREV` for specifying the source revision to use, be it `${AUTOREV}`, changelist number, p4date, or label, in preference to separate `SRC_URI` parameters to specify these. This change is more in-line with how the other fetchers work for source control systems. Recipes that fetch from Perforce will need to be updated to use `SRCREV` in place of specifying the source revision within `SRC_URI`.
- Some of BitBake's internal code structures for accessing the recipe cache needed to be changed to support the new multi-configuration functionality. These changes will affect external tools that use BitBake's tinfoil module. For information on these changes, see the changes made to the scripts supplied with OpenEmbedded-Core: [1](#) and [2](#).
- The task management code has been rewritten to avoid using ID indirection in order to improve performance. This change is unlikely to cause any problems for most users. However, the setscene verification function as pointed to by `BB_SETSCENE_VERIFY_FUNCTION` needed to change signature. Consequently, a new variable named `BB_SETSCENE_VERIFY_FUNCTION2` has been added allowing multiple versions of BitBake to work with suitably written metadata, which includes OpenEmbedded-Core and Poky. Anyone with custom BitBake task scheduler code might also need to update the code to handle the new structure.

### 24.10.11. Swabber has Been Removed¶

Swabber, a tool that was intended to detect host contamination in the build process, has been removed, as it has been unmaintained and unused for some time and was never particularly effective. The OpenEmbedded build system has since incorporated a number of mechanisms including enhanced QA checks that mean that there is less of a need for such a tool.

### 24.10.12. Removed Recipes¶

The following recipes have been removed:

- `augeas`: No longer needed and has been moved to `meta-oe`.
- `directfb`: Unmaintained and has been moved to `meta-oe`.
- `gcc`: Removed 4.9 version. Versions 5.4 and 6.2 are still present.
- `gnome-doc-utils`: No longer needed.
- `gtk-doc-stub`: Replaced by `gtk-doc`.

- `gtk-engines`: No longer needed and has been moved to `meta-gnome`.
- `gtk-sato-engine`: Became obsolete.
- `libglade`: No longer needed and has been moved to `meta-oe`.
- `libmad`: Unmaintained and functionally replaced by `libmpg123`. `libmad` has been moved to `meta-oe`.
- `libowl`: Became obsolete.
- `libxsettings-client`: No longer needed.
- `oh-puzzles`: Functionally replaced by `puzzles`.
- `oprofileui`: Became obsolete. OProfile has been largely supplanted by `perf`.
- `packagegroup-core-directfb.bb`: Removed.
- `core-image-directfb.bb`: Removed.
- `pointercal`: No longer needed and has been moved to `meta-oe`.
- `python-imaging`: No longer needed and moved to `meta-python`.
- `python-pyrex`: No longer needed and moved to `meta-python`.
- `sato-icon-theme`: Became obsolete.
- `swabber-native`: Swabber has been removed. See the [entry on Swabber](#).
- `tslib`: No longer needed and has been moved to `meta-oe`.
- `uclibc`: Removed in favor of `musl`.
- `xtscal`: No longer needed and moved to `meta-oe`.

#### 24.10.13. Removed Classes¶

The following classes have been removed:

- `distutils-native-base`: No longer needed.
- `distutils3-native-base`: No longer needed.
- `sdl`: Only set [DEPENDS](#) and [SECTION](#), which are better set within the recipe instead.
- `sip`: Mostly unused.
- `swabber`: See the [entry on Swabber](#).

#### 24.10.14. Minor Packaging Changes¶

The following minor packaging changes have occurred:

- `grub`: Split `grub-editenv` into its own package.
- `systemd`: Split container and vm related units into a new package, `systemd-container`.
- `util-linux`: Moved `prlimit` to a separate `util-linux-prlimit` package.

#### 24.10.15. Miscellaneous Changes¶

The following miscellaneous changes have occurred:

- `package_regex.inc`: Removed because the definitions `package_regex.inc` previously contained have been moved to their respective recipes.
- Both `devtool add` and `recipetool create` now use a fixed [SRCREV](#) by default when fetching from a Git repository. You can override this in either case to use `${AUTOREV}` instead by using the `-a` or `--autorev` command-line option.
- `distcc`: GTK+ UI is now disabled by default.
- `packagegroup-core-tools-testapps`: Removed Piglit.
- `image.bbclass`: Renamed `COMPRESS(ION)` to `CONVERSION`. This change means that `COMPRESSIONTYPES`, `COMPRESS_DEPENDS` and `COMPRESS_CMD` are deprecated in favor of `CONVERSIONTYPES`, `CONVERSION_DEPENDS` and `CONVERSION_CMD`. The `COMPRESS*` variable names will still work in the 2.2 release but metadata that does not need to be backwards-compatible should be changed to use the new names as the `COMPRESS*` ones will be removed in a future release.
- `gtk-doc`: A full version of `gtk-doc` is now made available. However, some old software might not be capable of using the current version of `gtk-doc` to build documentation. You need to change recipes that build such software so that they explicitly disable building documentation with `gtk-doc`.

### 24.11. Moving to the Yocto Project 2.3 Release¶

This section provides migration information for moving to the Yocto Project 2.3 Release from the prior release.

#### 24.11.1. Recipe-specific Sysroots¶

The OpenEmbedded build system now uses one sysroot per recipe to resolve long-standing issues with configuration script auto-detection of undeclared dependencies. Consequently, you might find that some of your previously written custom recipes are missing declared dependencies, particularly those dependencies that are incidentally built earlier in a typical build process and thus are already likely to be present in the shared sysroot in previous releases.

Consider the following:

- **Declare Build-Time Dependencies:** Because of this new feature, you must explicitly declare all build-time dependencies for your recipe. If you do not declare these dependencies, they are not populated into the sysroot for the recipe.
- **Specify Pre-Installation and Post-Installation Native Tool Dependencies:** You must specifically specify any special native tool dependencies of `pkg_preinst` and `pkg_postinst` scripts by using the `PACKAGE_WRITE_DEPS` variable. Specifying these dependencies ensures that these tools are available if these scripts need to be run on the build host during the `do_rootfs` task.

As an example, see the `dbus` recipe. You will see that this recipe has a `pkg_postinst` that calls `systemctl` if "systemd" is in `DISTRO_FEATURES`. In the example, `systemd-systemctl-native` is added to `PACKAGE_WRITE_DEPS`, which is also conditional on "systemd" being in `DISTRO_FEATURES`.

- **Examine Recipes that Use `SSTATEPOSTINSTFUNCS`:** You need to examine any recipe that uses `SSTATEPOSTINSTFUNCS` and determine steps to take.

Functions added to `SSTATEPOSTINSTFUNCS` are still called as they were in previous Yocto Project releases. However, since a separate sysroot is now being populated for every recipe and if existing functions being called through `SSTATEPOSTINSTFUNCS` are doing relocation, then you will need to change these to use a post-installation script that is installed by a function added to `SYSROOT_PREPROCESS_FUNCS`.

For an example, see the `pixbufcache` class in `meta/classes/` in the Yocto Project [Source Repositories](#).

## Note

The `SSTATEPOSTINSTFUNCS` variable itself is now deprecated in favor of the `do_populate_sysroot[postfuncs]` task. Consequently, if you do still have any function or functions that need to be called after the sysroot component is created for a recipe, then you would be well advised to take steps to use a post installation script as described previously. Taking these steps prepares your code for when `SSTATEPOSTINSTFUNCS` is removed in a future Yocto Project release.

- **Specify the Sysroot when Using Certain External Scripts:** Because the shared sysroot is now gone, the scripts `oe-find-native-sysroot` and `oe-run-native` have been changed such that you need to specify which recipe's `STAGING_DIR_NATIVE` is used.

## Note

You can find more information on how recipe-specific sysroots work in the "[staging.bbclass](#)" section.

### 24.11.2. PATH Variable¶

Within the environment used to run build tasks, the environment variable `PATH` is now sanitized such that the normal native binary paths (`/bin`, `/sbin`, `/usr/bin` and so forth) are removed and a directory containing symbolic links linking only to the binaries from the host mentioned in the `HOSTTOOLS` and `HOSTTOOLS_NONFATAL` variables is added to `PATH`.

Consequently, any native binaries provided by the host that you need to call needs to be in one of these two variables at the configuration level.

Alternatively, you can add a native recipe (i.e. `-native`) that provides the binary to the recipe's `DEPENDS` value.

## Note

`PATH` is not sanitized in the same way within `devshell`. If it were, you would have difficulty running host tools for development and debugging within the shell.

### 24.11.3. Changes to Scripts¶

The following changes to scripts took place:

- **`oe-find-native-sysroot`:** The usage for the `oe-find-native-sysroot` script has changed to the following:

```
$. oe-find-native-sysroot recipe
```

You must now supply a recipe for *recipe* as part of the command. Prior to the Yocto Project 3.0.1 release, it was not necessary to provide the script with the command.

- **`oe-run-native`:** The usage for the `oe-run-native` script has changed to the following:

```
$ oe-run-native native_recipe tool
```

You must supply the name of the native recipe and the tool you want to run as part of the command. Prior to the Yocto Project 3.0.1 release, it was not necessary to provide the native recipe with the command.



- **`cleanup-workdir`:** The `cleanup-workdir` script has been removed because the script was found to be deleting files it should not have, which lead to broken build trees. Rather than trying to delete portions of `TMPDIR` and getting it wrong, it is recommended that you delete `TMPDIR` and have it restored from shared state (`sstate`) on subsequent builds.
- **`wipe-sysroot`:** The `wipe-sysroot` script has been removed as it is no longer needed with recipe-specific sysroots.

#### 24.11.4. Changes to Functions¶

The previously deprecated `bb.data.getVar()`, `bb.data.setVar()`, and related functions have been removed in favor of `d.getVar()`, `d.setVar()`, and so forth.

You need to fix any references to these old functions.

#### 24.11.5. BitBake Changes¶

The following changes took place for BitBake:

- **BitBake's Graphical Dependency Explorer UI Replaced:** BitBake's graphical dependency explorer UI `depexp` was replaced by `taskexp` ("Task Explorer"), which provides a graphical way of exploring the `task-depends.dot` file. The data presented by Task Explorer is much more accurate than the data that was presented by `depexp`. Being able to visualize the data is an often requested feature as standard `*.dot` file viewers cannot usual cope with the size of the `task-depends.dot` file.
- **BitBake "-g" Output Changes:** The `package-depends.dot` and `pn-depends.dot` files as previously generated using the `bitbake -g` command have been removed. A `recipe-depends.dot` file is now generated as a collapsed version of `task-depends.dot` instead.

The reason for this change is because `package-depends.dot` and `pn-depends.dot` largely date back to a time before task-based execution and do not take into account task-level dependencies between recipes, which could be misleading.

- **Mirror Variable Splitting Changes:** Mirror variables including `MIRRORS`, `PREMIRRORS`, and `SSTATE_MIRRORS` can now separate values entirely with spaces. Consequently, you no longer need `"\\n"`. BitBake looks for pairs of values, which simplifies usage. There should be no change required to existing mirror variable values themselves.
- **The Subversion (SVN) Fetcher Uses an "ssh" Parameter and Not an "rsh" Parameter:** The SVN fetcher now takes an "ssh" parameter instead of an "rsh" parameter. This new optional parameter is used when the "protocol" parameter is set to "svn+ssh". You can only use the new parameter to specify the `ssh` program used by SVN. The SVN fetcher passes the new parameter through the `SVN_SSH` environment variable during the `do_fetch` task.

See the "Subversion (SVN) Fetcher (`svn://`)" section in the BitBake User Manual for additional information.

- **`BB_SETSCENE_VERIFY_FUNCTION` and `BB_SETSCENE_VERIFY_FUNCTION2` Removed:** Because the mechanism they were part of is no longer necessary with recipe-specific sysroots, the `BB_SETSCENE_VERIFY_FUNCTION` and `BB_SETSCENE_VERIFY_FUNCTION2` variables have been removed.

#### 24.11.6. Absolute Symbolic Links¶

Absolute symbolic links (symlinks) within staged files are no longer permitted and now trigger an error. Any explicit creation of symlinks can use the `lnr` script, which is a replacement for `ln -r`.

If the build scripts in the software that the recipe is building are creating a number of absolute symlinks that need to be corrected, you can inherit `relative_symlinks` within the recipe to turn those absolute symlinks into relative symlinks.

#### 24.11.7. GPLv2 Versions of GPLv3 Recipes Moved¶

Older GPLv2 versions of GPLv3 recipes have moved to a separate `meta-gplv2` layer.

If you use `INCOMPATIBLE_LICENSE` to exclude GPLv3 or set `PREFERRED_VERSION` to substitute a GPLv2 version of a GPLv3 recipe, then you must add the `meta-gplv2` layer to your configuration.

#### Note

You can find `meta-gplv2` layer in the OpenEmbedded layer index at <https://layers.openembedded.org/layerindex/branch/master/layer/meta-gplv2/>.

These relocated GPLv2 recipes do not receive the same level of maintenance as other core recipes. The recipes do not get security fixes and upstream no longer maintains them. In fact, the upstream community is actively hostile towards people that use the old versions of the recipes. Moving these recipes into a separate layer both makes the different needs of the recipes clearer and clearly identifies the number of these recipes.

#### Note

The long-term solution might be to move to BSD-licensed replacements of the GPLv3 components for those that need to exclude GPLv3-licensed components from the target system. This solution will be investigated for future Yocto Project releases.

#### 24.11.8. Package Management Changes¶

The following package management changes took place:

- Smart package manager is replaced by DNF package manager. Smart has become unmaintained upstream, is not ported to Python 3.x. Consequently, Smart needed to be replaced. DNF is the only feasible candidate.



The change in functionality is that the on-target runtime package management from remote package feeds is now done with a different tool that has a different set of command-line options. If you have scripts that call the tool directly, or use its API, they need to be fixed.

For more information, see the [DNF Documentation](#).

- Rpm 5.x is replaced with Rpm 4.x. This is done for two major reasons:
  - DNF is API-incompatible with Rpm 5.x and porting it and maintaining the port is non-trivial.
  - Rpm 5.x itself has limited maintenance upstream, and the Yocto Project is one of the very few remaining users.
- Berkeley DB 6.x is removed and Berkeley DB 5.x becomes the default:
  - Version 6.x of Berkeley DB has largely been rejected by the open source community due to its AGPLv3 license. As a result, most mainstream open source projects that require DB are still developed and tested with DB 5.x.
  - In OE-core, the only thing that was requiring DB 6.x was Rpm 5.x. Thus, no reason exists to continue carrying DB 6.x in OE-core.
- `createrepo` is replaced with `createrepo_c`.

`createrepo_c` is the current incarnation of the tool that generates remote repository metadata. It is written in C as compared to `createrepo`, which is written in Python. `createrepo_c` is faster and is maintained.

- Architecture-independent RPM packages are "noarch" instead of "all".
 

This change was made because too many places in DNF/RPM4 stack already make that assumption. Only the filenames and the architecture tag has changed. Nothing else has changed in OE-core system, particularly in the [allarch.bbclass](#) class.
- Signing of remote package feeds using `PACKAGE_FEED_SIGN` is not currently supported. This issue will be fully addressed in a future Yocto Project release. See [defect 11209](#) for more information on a solution to package feed signing with RPM in the Yocto Project 2.3 release.
- OPKG now uses the libsolv backend for resolving package dependencies by default. This is vastly superior to OPKG's internal ad-hoc solver that was previously used. This change does have a small impact on disk (around 500 KB) and memory footprint.

## Note

For further details on this change, see the [commit message](#).

### 24.11.9. Removed Recipes¶

The following recipes have been removed:

- **linux-yocto 4.8:** Version 4.8 has been removed. Versions 4.1 (LTSI), 4.4 (LTS), 4.9 (LTS/LTSI) and 4.10 are now present.
- **python-smarttpm:** Functionally replaced by `dnf`.
- **createrepo:** Replaced by the `createrepo-c` recipe.
- **rpmresolve:** No longer needed with the move to RPM 4 as RPM itself is used instead.
- **gststreamer:** Removed the GStreamer Git version recipes as they have been stale. 1.10.x recipes are still present.
- **alsa-conf-base:** Merged into `alsa-conf` since `libasound` depended on both. Essentially, no way existed to install only one of these.
- **tremor:** Moved to `meta-multimedia`. Fixed-integer Vorbis decoding is not needed by current hardware. Thus, GStreamer's `ivorbis` plugin has been disabled by default eliminating the need for the `tremor` recipe in [OE-Core](#).
- **gummiboot:** Replaced by `systemd-boot`.

### 24.11.10. Wic Changes¶

The following changes have been made to Wic:

## Note

For more information on Wic, see the ["Creating Partitioned Images Using Wic"](#) section in the Yocto Project Development Tasks Manual.

- **Default Output Directory Changed:** Wic's default output directory is now the current directory by default instead of the unusual `/var/tmp/wic`.

The `"-o"` and `"--outdir"` options remain unchanged and are used to specify your preferred output directory if you do not want to use the default directory.

- **fsimage Plug-in Removed:** The Wic `fsimage` plugin has been removed as it duplicates functionality of the `rawcopy` plugin.

### 24.11.11. QA Changes¶

The following QA checks have changed:

- **unsafe-references-in-binaries:** The `unsafe-references-in-binaries` QA check, which was disabled by default, has now been removed. This check was intended to detect binaries in `/bin` that link to libraries in `/usr/lib` and have the case where the user has `/usr` on a separate filesystem to `/`.

The removed QA check was buggy. Additionally, `/usr` residing on a separate partition from `/` is now a rare configuration. Consequently, `unsafe-references-in-binaries` was removed.

- **`file-rdeps`:** The `file-rdeps` QA check is now an error by default instead of a warning. Because it is an error instead of a warning, you need to address missing runtime dependencies.

For additional information, see the [insane](#) class and the "Errors and Warnings" section.

### 24.11.12. Miscellaneous Changes¶

The following miscellaneous changes have occurred:

- In this release, a number of recipes have been changed to ignore the `largefile` `DISTRO_FEATURES` item, enabling large file support unconditionally. This feature has always been enabled by default. Disabling the feature has not been widely tested.

#### Note

Future releases of the Yocto Project will remove entirely the ability to disable the `largefile` feature, which would make it unconditionally enabled everywhere.

- If the `DISTRO_VERSION` value contains the value of the `DATE` variable, which is the default between Poky releases, the `DATE` value is explicitly excluded from `/etc/issue` and `/etc/issue.net`, which is displayed at the login prompt, in order to avoid conflicts with Multilib enabled. Regardless, the `DATE` value is inaccurate if the `base-files` recipe is restored from shared state (`sstate`) rather than rebuilt.

If you need the build date recorded in `/etc/issue*` or anywhere else in your image, a better method is to define a post-processing function to do it and have the function called from `ROOTFS_POSTPROCESS_COMMAND`. Doing so ensures the value is always up-to-date with the created image.

- Dropbear's `init` script now disables DSA host keys by default. This change is in line with the `systemd` service file, which supports RSA keys only, and with recent versions of OpenSSH, which deprecates DSA host keys.
- The `buildhistory` class now correctly uses tabs as separators between all columns in `installed-package-sizes.txt` in order to aid import into other tools.
- The `USE_LDCONFIG` variable has been replaced with the "ldconfig" `DISTRO_FEATURES` feature. Distributions that previously set:

```
USE_LDCONFIG = "0"
```

should now instead use the following:

```
DISTRO_FEATURES_BACKFILL_CONSIDERED_append = " ldconfig"
```

- The default value of `COPYLEFT_LICENSE_INCLUDE` now includes all versions of AGPL licenses in addition to GPL and LGPL.

#### Note

The default list is not intended to be guaranteed as a complete safe list. You should seek legal advice based on what you are distributing if you are unsure.

- Kernel module packages are now suffixed with the kernel version in order to allow module packages from multiple kernel versions to co-exist on a target system. If you wish to return to the previous naming scheme that does not include the version suffix, use the following:

```
KERNEL_MODULE_PACKAGE_SUFFIX to ""
```

- Removal of `libtool *.la` files is now enabled by default. The `*.la` files are not actually needed on Linux and relocating them is an unnecessary burden.

If you need to preserve these `.la` files (e.g. in a custom distribution), you must change `INHERIT_DISTRO` such that "remove-libtool" is not included in the value.

- Extensible SDKs built for GCC 5+ now refuse to install on a distribution where the host GCC version is 4.8 or 4.9. This change resulted from the fact that the installation is known to fail due to the way the `uninative` shared state (`sstate`) package is built. See the [uninative](#) class for additional information.
- All native and nativesdk recipes now use a separate `DISTRO_FEATURES` value instead of sharing the value used by recipes for the target, in order to avoid unnecessary rebuilds.

The `DISTRO_FEATURES` for native recipes is `DISTRO_FEATURES_NATIVE` added to an intersection of `DISTRO_FEATURES` and `DISTRO_FEATURES_FILTER_NATIVE`.

For nativesdk recipes, the corresponding variables are `DISTRO_FEATURES_NATIVESDK` and `DISTRO_FEATURES_FILTER_NATIVESDK`.

- The `FILESDIR` variable, which was previously deprecated and rarely used, has now been removed. You should change any recipes that set `FILESDIR` to set `FILES_PATH` instead.
- The `MULTIMACH_HOST_SYS` variable has been removed as it is no longer needed with recipe-specific sysroots.

## 24.12. Moving to the Yocto Project 2.4 Release¶

This section provides migration information for moving to the Yocto Project 2.4 Release from the prior release.

### 24.12.1. Memory Resident Mode¶

A persistent mode is now available in BitBake's default operation, replacing its previous "memory resident mode" (i.e. `oe-init-build-env-memres`). Now you only need to set `BB_SERVER_TIMEOUT` to a timeout (in seconds) and BitBake's server stays resident for that amount of time between invocations. The `oe-init-build-env-memres` script has been removed since a separate environment setup script is no longer needed.

### 24.12.2. Packaging Changes¶

This section provides information about packaging changes that have occurred:

- **python3 Changes:**
  - The main "python3" package now brings in all of the standard Python 3 distribution rather than a subset. This behavior matches what is expected based on traditional Linux distributions. If you wish to install a subset of Python 3, specify `python-core` plus one or more of the individual packages that are still produced.
  - **python3:** The `bz2.py`, `lzma.py`, and `_compression.py` scripts have been moved from the `python3-misc` package to the `python3-compression` package.
- **binutils:** The `libbfd` library is now packaged in a separate "libbfd" package. This packaging saves space when certain tools (e.g. `perf`) are installed. In such cases, the tools only need `libbfd` rather than all the packages in `binutils`.
- **util-linux Changes:**
  - The `su` program is now packaged in a separate "util-linux-su" package, which is only built when "pam" is listed in the `DISTRO_FEATURES` variable. `util-linux` should not be installed unless it is needed because `su` is normally provided through the shadow file format. The main `util-linux` package has runtime dependencies (i.e. `RDEPENDS`) on the `util-linux-su` package when "pam" is in `DISTRO_FEATURES`.
  - The `switch_root` program is now packaged in a separate "util-linux-switch-root" package for small initramfs images that do not need the whole `util-linux` package or the `busybox` binary, which are both much larger than `switch_root`. The main `util-linux` package has a recommended runtime dependency (i.e. `RRECOMMENDS`) on the `util-linux-switch-root` package.
  - The `ionice` program is now packaged in a separate "util-linux-ionice" package. The main `util-linux` package has a recommended runtime dependency (i.e. `RRECOMMENDS`) on the `util-linux-ionice` package.
- **initscripts:** The `sushell` program is now packaged in a separate "initscripts-sushell" package. This packaging change allows systems to pull `sushell` in when `selinux` is enabled. The change also eliminates needing to pull in the entire `initscripts` package. The main `initscripts` package has a runtime dependency (i.e. `RDEPENDS`) on the `sushell` package when "selinux" is in `DISTRO_FEATURES`.
- **glib-2.0:** The `glib-2.0` package now has a recommended runtime dependency (i.e. `RRECOMMENDS`) on the `shared-mime-info` package, since large portions of GIO are not useful without the MIME database. You can remove the dependency by using the `BAD_RECOMMENDATIONS` variable if `shared-mime-info` is too large and is not required.
- **Go Standard Runtime:** The Go standard runtime has been split out from the main `go` recipe into a separate `go-runtime` recipe.

### 24.12.3. Removed Recipes¶

The following recipes have been removed:

- **acpittests:** This recipe is not maintained.
- **autogen-native:** No longer required by Grub, `oe-core`, or `meta-oe`.
- **bdwgc:** Nothing in OpenEmbedded-Core requires this recipe. It has moved to `meta-oe`.
- **byacc:** This recipe was only needed by `rpm 5.x` and has moved to `meta-oe`.
- **gcc (5.4):** The 5.4 series dropped the recipe in favor of 6.3 / 7.2.
- **gnome-common:** Deprecated upstream and no longer needed.
- **go-bootstrap-native:** Go 1.9 does its own bootstrapping so this recipe has been removed.
- **guile:** This recipe was only needed by `autogen-native` and `remake`. The recipe is no longer needed by either of these programs.
- **libclass-isa-perl:** This recipe was previously needed for LSB 4, no longer needed.
- **libdumpvalue-perl:** This recipe was previously needed for LSB 4, no longer needed.
- **libenv-perl:** This recipe was previously needed for LSB 4, no longer needed.
- **libfile-checktree-perl:** This recipe was previously needed for LSB 4, no longer needed.
- **libi18n-collate-perl:** This recipe was previously needed for LSB 4, no longer needed.
- **libiconv:** This recipe was only needed for `uclibc`, which was removed in the previous release. `glibc` and `musl` have their own implementations. `meta-mingw` still needs `libiconv`, so it has been moved to `meta-mingw`.
- **libpng12:** This recipe was previously needed for LSB. The current `libpng` is 1.6.x.
- **libpod-plainer-perl:** This recipe was previously needed for LSB 4, no longer needed.

- **linux-yocto (4.1):** This recipe was removed in favor of 4.4, 4.9, 4.10 and 4.12.
- **mailx:** This recipe was previously only needed for LSB compatibility, and upstream is defunct.
- **mesa (git version only):** The git version recipe was stale with respect to the release version.
- **ofono (git version only):** The git version recipe was stale with respect to the release version.
- **portmap:** This recipe is obsolete and is superseded by `rpcbind`.
- **python3-pygpme:** This recipe is old and unmaintained. It was previously required by `dnf`, which has switched to official `gpme` Python bindings.
- **python-async:** This recipe has been removed in favor of the Python 3 version.
- **python-gitdb:** This recipe has been removed in favor of the Python 3 version.
- **python-git:** This recipe was removed in favor of the Python 3 version.
- **python-mako:** This recipe was removed in favor of the Python 3 version.
- **python-pexpect:** This recipe was removed in favor of the Python 3 version.
- **python-ptyprocess:** This recipe was removed in favor of Python the 3 version.
- **python-pycurl:** Nothing is using this recipe in OpenEmbedded-Core (i.e. `meta-oe`).
- **python-six:** This recipe was removed in favor of the Python 3 version.
- **python-smmap:** This recipe was removed in favor of the Python 3 version.
- **remake:** Using `remake` as the provider of `virtual/make` is broken. Consequently, this recipe is not needed in OpenEmbedded-Core.

#### 24.12.4. Kernel Device Tree Move¶

Kernel Device Tree support is now easier to enable in a kernel recipe. The Device Tree code has moved to a `kernel-devicetree` class. Functionality is automatically enabled for any recipe that inherits the `kernel` class and sets the `KERNEL_DEVICETREE` variable. The previous mechanism for doing this, `meta/recipes-kernel/linux/linux-dtb.inc`, is still available to avoid breakage, but triggers a deprecation warning. Future releases of the Yocto Project will remove `meta/recipes-kernel/linux/linux-dtb.inc`. It is advisable to remove any `require` statements that request `meta/recipes-kernel/linux/linux-dtb.inc` from any custom kernel recipes you might have. This will avoid breakage in post 2.4 releases.

#### 24.12.5. Package QA Changes¶

The following package QA changes took place:

- The "unsafe-references-in-scripts" QA check has been removed.
- If you refer to `${COREBASE}/LICENSE` within `LIC_FILES_CHKSUM` you receive a warning because this file is a description of the license for OE-Core. Use `${COMMON_LICENSE_DIR}/MIT` if your recipe is MIT-licensed and you cannot use the preferred method of referring to a file within the source tree.

#### 24.12.6. README File Changes¶

The following are changes to README files:

- The main Poky README file has been moved to the `meta-poky` layer and has been renamed `README.poky`. A symlink has been created so that references to the old location work.
- The `README.hardware` file has been moved to `meta-yocto-bsp`. A symlink has been created so that references to the old location work.
- A `README.qemu` file has been created with coverage of the `qemu*` machines.

#### 24.12.7. Miscellaneous Changes¶

The following are additional changes:

- The `ROOTFS_PKGMANAGE_BOOTSTRAP` variable and any references to it have been removed. You should remove this variable from any custom recipes.
- The `meta-yocto` directory has been removed.

### Note

In the Yocto Project 2.1 release `meta-yocto` was renamed to `meta-poky` and the `meta-yocto` subdirectory remained to avoid breaking existing configurations.

- The `maintainers.inc` file, which tracks maintainers by listing a primary person responsible for each recipe in OE-Core, has been moved from `meta-poky` to OE-Core (i.e. from `meta-poky/conf/distro/include` to `meta/conf/distro/include`).
- The `buildhistory` class now makes a single commit per build rather than one commit per subdirectory in the repository. This behavior assumes the commits are enabled with `BUILDHISTORY_COMMIT = "1"`, which is typical. Previously, the `buildhistory` class made one commit per subdirectory in the repository in order to make it easier to see the changes for a

particular subdirectory. To view a particular change, specify that subdirectory as the last parameter on the `git show` or `git diff` commands.

- The `x86-base.inc` file, which is included by all x86-based machine configurations, now sets `IMAGE_FSTYPES` using `?` to "live" rather than appending with `+=`. This change makes the default easier to override.
- BitBake fires multiple "BuildStarted" events when multiconfig is enabled (one per configuration). For more information, see the "Events" section in the BitBake User Manual.
- By default, the `security_flags.inc` file sets a `GCCPIE` variable with an option to enable Position Independent Executables (PIE) within `gcc`. Enabling PIE in the GNU C Compiler (GCC), makes Return Oriented Programming (ROP) attacks much more difficult to execute.
- OE-Core now provides a `bitbake-layers` plugin that implements a "create-layer" subcommand. The implementation of this subcommand has resulted in the `yocto-layer` script being deprecated and will likely be removed in the next Yocto Project release.
- The `vmdk`, `vdi`, and `qcow2` image file types are now used in conjunction with the "wic" image type through `CONVERSION_CMD`. Consequently, the equivalent image types are now `wic.vmdk`, `wic.vdi`, and `wic.qcow2`, respectively.
- `do_image_<type>[depends]` has replaced `IMAGE_DEPENDS_<type>`. If you have your own classes that implement custom image types, then you need to update them.
- OpenSSL 1.1 has been introduced. However, the default is still 1.0.x through the `PREFERRED_VERSION` variable. This preference is set is due to the remaining compatibility issues with other software. The `PROVIDES` variable in the `openssl 1.0` recipe now includes "openssl10" as a marker that can be used in `DEPENDS` within recipes that build software that still depend on OpenSSL 1.0.
- To ensure consistent behavior, BitBake's "-r" and "-R" options (i.e. prefile and postfile), which are used to read or post-read additional configuration files from the command line, now only affect the current BitBake command. Before these BitBake changes, these options would "stick" for future executions.

## 24.13. Moving to the Yocto Project 2.5 Release¶

This section provides migration information for moving to the Yocto Project 2.5 Release from the prior release.

### 24.13.1. Packaging Changes¶

This section provides information about packaging changes that have occurred:

- **bind-libs:** The libraries packaged by the `bind` recipe are in a separate `bind-libs` package.
- **libfm-gtk:** The `libfm` GTK+ bindings are split into a separate `libfm-gtk` package.
- **flex-libfl:** The `flex` recipe splits out `libfl` into a separate `flex-libfl` package to avoid too many dependencies being pulled in where only the library is needed.
- **grub-efi:** The `grub-efi` configuration is split into a separate `grub-bootconf` recipe. However, the dependency relationship from `grub-efi` is through a virtual/`grub-bootconf` provider making it possible to have your own recipe provide the dependency. Alternatively, you can use a BitBake append file to bring the configuration back into the `grub-efi` recipe.
- **armv7a Legacy Package Feed Support:** Legacy support is removed for transitioning from `armv7a` to `armv7a-vfp-neon` in package feeds, which was previously enabled by setting `PKGARCHCOMPAT_ARMV7A`. This transition occurred in 2011 and active package feeds should by now be updated to the new naming.

### 24.13.2. Removed Recipes¶

The following recipes have been removed:

- **gcc:** The version 6.4 recipes are replaced by 7.x.
- **gst-player:** Renamed to `gst-examples` as per upstream.
- **hostap-utils:** This software package is obsolete.
- **latencytop:** This recipe is no longer maintained upstream. The last release was in 2009.
- **libpfm4:** The only file that requires this recipe is `oprofile`, which has been removed.
- **linux-yocto:** The version 4.4, 4.9, and 4.10 recipes have been removed. Versions 4.12, 4.14, and 4.15 remain.
- **man:** This recipe has been replaced by modern `man-db`
- **mkelfimage:** This tool has been removed in the upstream coreboot project, and is no longer needed with the removal of the ELF image type.
- **nativesdk-postinst-intercept:** This recipe is not maintained.
- **neon:** This software package is no longer maintained upstream and is no longer needed by anything in OpenEmbedded-Core.
- **oprofile:** The functionality of this recipe is replaced by `perf` and keeping compatibility on an ongoing basis with `musl` is difficult.
- **pax:** This software package is obsolete.
- **stat:** This software package is not maintained upstream. `coreutils` provides a modern `stat` binary.
- **zisofs-tools-native:** This recipe is no longer needed because the compressed ISO image feature has been removed.

### 24.13.3. Scripts and Tools Changes¶

The following are changes to scripts and tools:

- **yocto-bsp, yocto-kernel, and yocto-layer:** The `yocto-bsp`, `yocto-kernel`, and `yocto-layer` scripts previously shipped with poky but not in OpenEmbedded-Core have been removed. These scripts are not maintained and are outdated. In many cases, they are also limited in scope. The `bitbake-layers create-layer` command is a direct replacement for `yocto-layer`. See the documentation to create a BSP or kernel recipe in the "[BSP Kernel Recipe Example](#)" section.
- **devtool finish:** `devtool finish` now exits with an error if there are uncommitted changes or a rebase/am in progress in the recipe's source repository. If this error occurs, there might be uncommitted changes that will not be included in updates to the patches applied by the recipe. A `-f/--force` option is provided for situations that the uncommitted changes are inconsequential and you want to proceed regardless.
- **scripts/oe-setup-rpmrepo script:** The functionality of `scripts/oe-setup-rpmrepo` is replaced by `bitbake package-index`.
- **scripts/test-dependencies.sh script:** The script is largely made obsolete by the recipe-specific sysroots functionality introduced in the previous release.

### 24.13.4. BitBake Changes¶

The following are BitBake changes:

- The `--runall` option has changed. There are two different behaviors people might want:
  - **Behavior A:** For a given target (or set of targets) look through the task graph and run task X only if it is present and will be built.
  - **Behavior B:** For a given target (or set of targets) look through the task graph and run task X if any recipe in the taskgraph has such a target, even if it is not in the original task graph.

The `--runall` option now performs "Behavior B". Previously `--runall` behaved like "Behavior A". A `--runonly` option has been added to retain the ability to perform "Behavior A".

- Several explicit "run this task for all recipes in the dependency tree" tasks have been removed (e.g. `fetchall`, `checkuriall`, and the `*all` tasks provided by the `distrodata` and `archiver` classes). There is a BitBake option to complete this for any arbitrary task. For example:

```
bitbake <target> -c fetchall
```

should now be replaced with:

```
bitbake <target> --runall=fetch
```

### 24.13.5. Python and Python 3 Changes¶

The following are auto-packaging changes to Python and Python 3:

The script-managed `python-*manifest.inc` files that were previously used to generate Python and Python 3 packages have been replaced with a JSON-based file that is easier to read and maintain. A new task is available for maintainers of the Python recipes to update the JSON file when upgrading to new Python versions. You can now edit the file directly instead of having to edit a script and run it to update the file.

One particular change to note is that the Python recipes no longer have build-time provides for their packages. This assumes `python-foo` is one of the packages provided by the Python recipe. You can no longer run `bitbake python-foo` or have a `DEPENDS` on `python-foo`, but doing either of the following causes the package to work as expected:

```
IMAGE_INSTALL_append = " python-foo"
```

or

```
RDEPENDS_${PN} = "python-foo"
```

The earlier build-time provides behavior was a quirk of the way the Python manifest file was created. For more information on this change please see [this commit](#).

### 24.13.6. Miscellaneous Changes¶

The following are additional changes:

- The `kernel` class supports building packages for multiple kernels. If your kernel recipe or `.bbappend` file mentions packaging at all, you should replace references to the kernel in package names with `${KERNEL_PACKAGE_NAME}`. For example, if you disable automatic installation of the kernel image using `RDEPENDS kernel-base = ""` you can avoid warnings using `RDEPENDS_${KERNEL_PACKAGE_NAME}-base = ""` instead.
- The `buildhistory` class commits changes to the repository by default so you no longer need to set `BUILDHISTORY_COMMIT = "1"`. If you want to disable commits you need to set `BUILDHISTORY_COMMIT = "0"` in your configuration.
- The `beaglebone` reference machine has been renamed to `beaglebone-yocto`. The `beaglebone-yocto` BSP is a reference implementation using only mainline components available in OpenEmbedded-Core and `meta-yocto-bsp`, whereas Texas Instruments maintains a full-featured BSP in the `meta-ti` layer. This rename avoids the previous name clash that existed between the two BSPs.
- The `update-alternatives` class no longer works with SysV `init` scripts because this usage has been problematic. Also, the `syslogd` recipe no longer uses `update-alternatives` because it is incompatible with other implementations.



- By default, the `cmake` class uses `ninja` instead of `make` for building. This improves build performance. If a recipe is broken with `ninja`, then the recipe can set `OECMAKE_GENERATOR = "Unix Makefiles"` to change back to `make`.
- The previously deprecated `base_*` functions have been removed in favor of their replacements in `meta/lib/oe` and `bitbake/lib/bb`. These are typically used from recipes and classes. Any references to the old functions must be updated. The following table shows the removed functions and their replacements:

| Removed                                   | Replacement                                                                        |
|-------------------------------------------|------------------------------------------------------------------------------------|
| =====                                     | =====                                                                              |
| <code>base_path_join()</code>             | <code>oe.path.join()</code>                                                        |
| <code>base_path_relative()</code>         | <code>oe.path.relative()</code>                                                    |
| <code>base_path_out()</code>              | <code>oe.path.format_display()</code>                                              |
| <code>base_read_file()</code>             | <code>oe.utils.read_file()</code>                                                  |
| <code>base_ifelse()</code>                | <code>oe.utils.ifelse()</code>                                                     |
| <code>base_conditional()</code>           | <code>oe.utils.conditional()</code>                                                |
| <code>base_less_or_equal()</code>         | <code>oe.utils.less_or_equal()</code>                                              |
| <code>base_version_less_or_equal()</code> | <code>oe.utils.version_less_or_equal()</code>                                      |
| <code>base_contains()</code>              | <code>bb.utils.contains()</code>                                                   |
| <code>base_both_contain()</code>          | <code>oe.utils.both_contain()</code>                                               |
| <code>base_prune_suffix()</code>          | <code>oe.utils.prune_suffix()</code>                                               |
| <code>oe_filter()</code>                  | <code>oe.utils.str_filter()</code>                                                 |
| <code>oe_filter_out()</code>              | <code>oe.utils.str_filter_out()</code> (or use the <code>_remove</code> operator). |

- Using `exit 1` to explicitly defer a postinstall script until first boot is now deprecated since it is not an obvious mechanism and can mask actual errors. If you want to explicitly defer a postinstall to first boot on the target rather than at `rootfs` creation time, use `pkg_postinst_ontarget()` or call `postinst-intercepts defer_to_first_boot` from `pkg_postinst()`. Any failure of a `pkg_postinst()` script (including `exit 1`) will trigger a warning during `do_rootfs`.

For more information, see the "[Post-Installation Scripts](#)" section in the Yocto Project Development Tasks Manual.

- The `elf` image type has been removed. This image type was removed because the `mkelfimage` tool that was required to create it is no longer provided by `coreboot` upstream and required updating every time `binutils` updated.
- Support for `.iso` image compression (previously enabled through `COMPRESSISO = "1"`) has been removed. The userspace tools (`zisoofs-tools`) are unmaintained and `squashfs` provides better performance and compression. In order to build a live image with `squashfs+lz4` compression enabled you should now set `LIVE_ROOTFS_TYPE = "squashfs-lz4"` and ensure that `live` is in `IMAGE_FSTYPES`.
- Recipes with an unconditional dependency on `libpam` are only buildable with `pam` in `DISTRO_FEATURES`. If the dependency is truly optional then it is recommended that the dependency be conditional upon `pam` being in `DISTRO_FEATURES`.
- For EFI-based machines, the bootloader (`grub-efi` by default) is installed into the image at `/boot`. `Wic` can be used to split the bootloader into separate boot and `rootfs` partitions if necessary.
- Patches whose context does not match exactly (i.e. where patch reports "fuzz" when applying) will generate a warning. For an example of this see [this commit](#).
- Layers are expected to set `LAYERSERIES_COMPAT_layername` to match the version(s) of OpenEmbedded-Core they are compatible with. This is specified as codenames using spaces to separate multiple values (e.g. "rocko sumo"). If a layer does not set `LAYERSERIES_COMPAT_layername`, a warning will be shown. If a layer sets a value that does not include the current version ("sumo" for the 2.5 release), then an error will be produced.
- The `TZ` environment variable is set to "UTC" within the build environment in order to fix reproducibility problems in some recipes.

## 24.14. Moving to the Yocto Project 2.6 Release¶

This section provides migration information for moving to the Yocto Project 2.6 Release from the prior release.

### 24.14.1. GCC 8.2 is Now Used by Default¶

The GNU Compiler Collection version 8.2 is now used by default for compilation. For more information on what has changed in the GCC 8.x release, see <https://gcc.gnu.org/gcc-8/changes.html>.

If you still need to compile with version 7.x, GCC 7.3 is also provided. You can select this version by setting the and can be selected by setting the `GCCVERSION` variable to "7.%" in your configuration.

### 24.14.2. Removed Recipes¶

The following recipes have been removed:

**beecrypt**: No longer needed since moving to RPM 4.  
**bigreqsproto**: Replaced by `xorgproto`.  
**calibrateproto**: Removed in favor of `xinput`.  
**compositeproto**: Replaced by `xorgproto`.  
**damageproto**: Replaced by `xorgproto`.  
**dmxproto**: Replaced by `xorgproto`.  
**dri2proto**: Replaced by `xorgproto`.  
**dri3proto**: Replaced by `xorgproto`.  
**eee-acpi-scripts**: Became obsolete.  
**fixesproto**: Replaced by `xorgproto`.  
**fontspROTO**: Replaced by `xorgproto`.  
**fstests**: Became obsolete.  
**gccmakedep**: No longer used.  
**glproto**: Replaced by `xorgproto`.  
**anome-desktop3**: No longer needed. This recipe has moved to `meta-oe`.

**systemd-udev**: No longer needed. This recipe has moved to `udev`.

**icon-naming-utils**: No longer used since the Sato theme was removed in 2016.

**inputproto**: Replaced by `xorgproto`.

**kbproto**: Replaced by `xorgproto`.

**libusb-compat**: Became obsolete.

**libuser**: Became obsolete.

**libnfsidmap**: No longer an external requirement since `nfs-utils` 2.2.1. `libnfsidmap` is now integrated.

**libxcalibrate**: No longer needed with `xinput`.

**mktemp**: Became obsolete. The `mktemp` command is provided by both `busybox` and `coreutils`.

**oss-p-uuid**: Is not being maintained and has mostly been replaced by `uuid.h` in `util-linux`.

**pax-utils**: No longer needed. Previous QA tests that did use this recipe are now done at build time.

**pcmciautils**: Became obsolete.

**pixz**: No longer needed. XZ now supports multi-threaded compression.

**presentproto**: Replaced by `xorgproto`.

**randrproto**: Replaced by `xorgproto`.

**recordproto**: Replaced by `xorgproto`.

**renderproto**: Replaced by `xorgproto`.

**resourceproto**: Replaced by `xorgproto`.

**scrnsaverproto**: Replaced by `xorgproto`.

**trace-cmd**: Became obsolete. `perf` replaced this recipe's functionality.

**videoproto**: Replaced by `xorgproto`.

**wireless-tools**: Became obsolete. Superseded by `iw`.

**xcmiscproto**: Replaced by `xorgproto`.

**xextproto**: Replaced by `xorgproto`.

**xf86dgaproto**: Replaced by `xorgproto`.

**xf86driproto**: Replaced by `xorgproto`.

**xf86miscproto**: Replaced by `xorgproto`.

**xf86-video-omapfb**: Became obsolete. Use kernel modesetting driver instead.

**xf86-video-omap**: Became obsolete. Use kernel modesetting driver instead.

**xf86vidmodeproto**: Replaced by `xorgproto`.

**xineramaproto**: Replaced by `xorgproto`.

**xproto**: Replaced by `xorgproto`.

**yasm**: No longer needed since previous usages are now satisfied by `nasm`.

### 24.14.3. Packaging Changes¶

The following packaging changes have been made:

- **cmake**: `cmake.m4` and `toolchain` files have been moved to the main package.
- **iptables**: The `iptables` modules have been split into separate packages.
- **alsa-lib**: `libasound` is now in the main `alsa-lib` package instead of `libasound`.
- **glibc**: `libnss-db` is now in its own package along with a `/var/db/makedbs.sh` script to update databases.
- **python and python3**: The main package has been removed from the recipe. You must install specific packages or `python-modules` / `python3-modules` for everything.
- **systemtap**: Moved `systemtap-exporter` into its own package.

### 24.14.4. XOrg Protocol dependencies¶

The "\*proto" upstream repositories have been combined into one "xorgproto" repository. Thus, the corresponding recipes have also been combined into a single `xorgproto` recipe. Any recipes that depend upon the older \*proto recipes need to be changed to depend on the newer `xorgproto` recipe instead.

For names of recipes removed because of this repository change, see the [Removed Recipes](#) section.

### 24.14.5. distutils and distutils3 Now Prevent Fetching Dependencies During the do\_configure Task¶

Previously, it was possible for Python recipes that inherited the `distutils` and `distutils3` classes to fetch code during the `do_configure` task to satisfy dependencies mentioned in `setup.py` if those dependencies were not provided in the sysroot (i.e. recipes providing the dependencies were missing from `DEPENDS`).

#### Note

This change affects classes beyond just the two mentioned (i.e. `distutils` and `distutils3`). Any recipe that inherits `distutils*` classes are affected. For example, the `setuptools` and `setuptools3` recipes are affected since they inherit the `distutils*` classes.

Fetching these types of dependencies that are not provided in the sysroot negatively affects the ability to reproduce builds. This type of fetching is now explicitly disabled. Consequently, any missing dependencies in Python recipes that use these classes now result in an error during the `do_configure` task.

### 24.14.6. linux-yocto Configuration Audit Issues Now Correctly Reported¶



Due to a bug, the kernel configuration audit functionality was not writing out any resulting warnings during the build. This issue is now corrected. You might notice these warnings now if you have a custom kernel configuration with a `linux-yocto` style kernel recipe.

### 24.14.7. Image/Kernel Artifact Naming Changes¶

The following changes have been made:

- Name variables (e.g. `IMAGE_NAME`) use a new `IMAGE_VERSION_SUFFIX` variable instead of `DATETIME`. Using `IMAGE_VERSION_SUFFIX` allows easier and more direct changes.

The `IMAGE_VERSION_SUFFIX` variable is set in the `bitbake.conf` configuration file as follows:

```
IMAGE_VERSION_SUFFIX = "-${DATETIME}"
```

- Several variables have changed names for consistency:

| Old Variable Name                        | New Variable Name                     |
|------------------------------------------|---------------------------------------|
| <code>KERNEL_IMAGE_BASE_NAME</code>      | <code>KERNEL_IMAGE_NAME</code>        |
| <code>KERNEL_IMAGE_SYMLINK_NAME</code>   | <code>KERNEL_IMAGE_LINK_NAME</code>   |
| <code>MODULE_TARBALL_BASE_NAME</code>    | <code>MODULE_TARBALL_NAME</code>      |
| <code>MODULE_TARBALL_SYMLINK_NAME</code> | <code>MODULE_TARBALL_LINK_NAME</code> |
| <code>INITRAMFS_BASE_NAME</code>         | <code>INITRAMFS_NAME</code>           |

- The `MODULE_IMAGE_BASE_NAME` variable has been removed. The module tarball name is now controlled directly with the `MODULE_TARBALL_NAME` variable.
- The `KERNEL_DTB_NAME` and `KERNEL_DTB_LINK_NAME` variables have been introduced to control kernel Device Tree Binary (DTB) artifact names instead of mangling `KERNEL_IMAGE_*` variables.
- The `KERNEL_FIT_NAME` and `KERNEL_FIT_LINK_NAME` variables have been introduced to specify the name of flattened image tree (FIT) kernel images similar to other deployed artifacts.
- The `MODULE_TARBALL_NAME` and `MODULE_TARBALL_LINK_NAME` variable values no longer include the "module-" prefix or ".tgz" suffix. These parts are now hardcoded so that the values are consistent with other artifact naming variables.
- Added the `INITRAMFS_LINK_NAME` variable so that the symlink can be controlled similarly to other artifact types.
- `INITRAMFS_NAME` now uses `"${PKGGE}-${PKGVS}-${PKGR}-${MACHINE}-${IMAGE_VERSION_SUFFIX}"` instead of `"${PV}-${PR}-${MACHINE}-${DATETIME}"`, which makes it consistent with other variables.

### 24.14.8. SERIAL\_CONSOLE Deprecated¶

The `SERIAL_CONSOLE` variable has been functionally replaced by the `SERIAL_CONSOLES` variable for some time. With the Yocto Project 2.6 release, `SERIAL_CONSOLE` has been officially deprecated.

`SERIAL_CONSOLE` will continue to work as before for the 2.6 release. However, for the sake of future compatibility, it is recommended that you replace all instances of `SERIAL_CONSOLE` with `SERIAL_CONSOLES`.

#### Note

The only difference in usage is that `SERIAL_CONSOLES` expects entries to be separated using semicolons as compared to `SERIAL_CONSOLE`, which expects spaces.

### 24.14.9. Configure Script Reports Unknown Options as Errors¶

If the configure script reports an unknown option, this now triggers a QA error instead of a warning. Any recipes that previously got away with specifying such unknown options now need to be fixed.

### 24.14.10. Override Changes¶

The following changes have occurred:

- The `virtclass-native` and `virtclass-nativesdk` Overrides Have Been Removed:** The `virtclass-native` and `virtclass-nativesdk` overrides have been deprecated since 2012 in favor of `class-native` and `class-nativesdk`, respectively. Both `virtclass-native` and `virtclass-nativesdk` are now dropped.

#### Note

The `virtclass-multilib` overrides for multilib are still valid.

- The `forcevariable` Override Now Has a Higher Priority Than `libc` Overrides:** The `forcevariable` override is documented to be the highest priority override. However, due to a long-standing quirk of how `OVERRIDES` is set, the `libc` overrides (e.g. `libc-glibc`, `libc-musl`, and so forth) erroneously had a higher priority. This issue is now corrected.

It is likely this change will not cause any problems. However, it is possible with some unusual configurations that you might see a change in behavior if you were relying on the previous behavior. Be sure to check how you use `forcevariable` and `libc-*` overrides in your custom layers and configuration files to ensure they make sense.

- The `build-${BUILD_OS}` Override Has Been Removed:** The `build-${BUILD_OS}`, which is typically `build-linux`, override has been removed because building on a host operating system other than a recent version of Linux is neither

supported nor recommended. Dropping the override avoids giving the impression that other host operating systems might be supported.

- The `"_remove"` operator now preserves whitespace. Consequently, when specifying list items to remove, be aware that leading and trailing whitespace resulting from the removal is retained.

See the ["Removal \(Override Style Syntax\)"](#) section in the BitBake User Manual for a detailed example.

#### 24.14.11. `systemd` Configuration is Now Split Into `systemd-conf`

The configuration for the `systemd` recipe has been moved into a `system-conf` recipe. Moving this configuration to a separate recipe avoids the `systemd` recipe from becoming machine-specific for cases where machine-specific configurations need to be applied (e.g. for `qemu*` machines).

Currently, the new recipe packages the following files:

```
{sysconfdir}/machine-id
{sysconfdir}/systemd/coredump.conf
{sysconfdir}/systemd/journald.conf
{sysconfdir}/systemd/logind.conf
{sysconfdir}/systemd/system.conf
{sysconfdir}/systemd/user.conf
```

If you previously used `bbappend` files to append the `systemd` recipe to change any of the listed files, you must do so for the `systemd-conf` recipe instead.

#### 24.14.12. Automatic Testing Changes

This section provides information about automatic testing changes:

- **`TEST_IMAGE` Variable Removed:** Prior to this release, you set the `TEST_IMAGE` variable to "1" to enable automatic testing for successfully built images. The `TEST_IMAGE` variable no longer exists and has been replaced by the `TESTIMAGE_AUTO` variable.
- **Inheriting the `testimage` and `testsdk` Classes:** Best practices now dictate that you use the `IMAGE_CLASSES` variable rather than the `INHERIT` variable when you inherit the `testimage` and `testsdk` classes used for automatic testing.

#### 24.14.13. OpenSSL Changes

`OpenSSL` has been upgraded from 1.0 to 1.1. By default, this upgrade could cause problems for recipes that have both versions in their dependency chains. The problem is that both versions cannot be installed together at build time.

#### Note

It is possible to have both versions of the library at runtime.

#### 24.14.14. BitBake Changes

The server logfile `bitbake-cookerdaemon.log` is now always placed in the [Build Directory](#) instead of the current directory.

#### 24.14.15. Security Changes

The Poky distribution now uses security compiler flags by default. Inclusion of these flags could cause new failures due to stricter checking for various potential security issues in code.

#### 24.14.16. Post Installation Changes

You must explicitly mark post installs to defer to the target. If you want to explicitly defer a postinstall to first boot on the target rather than at rootfs creation time, use `pkg_postinst_ontarget()` or call `postinst-intercepts defer_to_first_boot from pkg_postinst()`. Any failure of a `pkg_postinst()` script (including exit 1) triggers an error during the `do_rootfs` task.

For more information on post-installation behavior, see the ["Post-Installation Scripts"](#) section in the Yocto Project Development Tasks Manual.

#### 24.14.17. Python 3 Profile-Guided Optimization

The `python3` recipe now enables profile-guided optimization. Using this optimization requires a little extra build time in exchange for improved performance on the target at runtime. Additionally, the optimization is only enabled if the current `MACHINE` has support for user-mode emulation in QEMU (i.e. "qemu-usermode" is in `MACHINE_FEATURES`, which it is by default).

If you wish to disable Python profile-guided optimization regardless of the value of `MACHINE_FEATURES`, then ensure that `PACKAGECONFIG` for the `python3` recipe does not contain "pgo". You could accomplish the latter using the following at the configuration level:

```
PACKAGECONFIG_remove_pn-python3 = "pgo"
```

Alternatively, you can set `PACKAGECONFIG` using an append file for the `python3` recipe.

#### 24.14.18. Miscellaneous Changes

The following miscellaneous changes occurred:

- Default to using the Thumb-2 instruction set for armv7a and above. If you have any custom recipes that build software that needs to be built with the ARM instruction set, change the recipe to set the instruction set as follows:

```
ARM_INSTRUCTION_SET = "arm"
```

- `run-postinsts` no longer uses `/etc/*-postinsts` for `dpkg/opkg` in favor of built-in postinst support. RPM behavior remains unchanged.
- The `NOISO` and `NOHDD` variables are no longer used. You now control building `*.iso` and `*.hddimg` image types directly by using the `IMAGE_FSTYPES` variable.
- The `scripts/contrib/mkefidisk.sh` has been removed in favor of `Wic`.
- `kernel-modules` has been removed from `RRECOMMENDS` for `qemumips` and `qemumips64` machines. Removal also impacts the `x86-base.inc` file.

### Note

`genericx86` and `genericx86-64` retain `kernel-modules` as part of the `RRECOMMENDS` variable setting.

- The `LGPLv2_WHITELIST_GPL-3.0` variable has been removed. If you are setting this variable in your configuration, set or append it to the `WHITELIST_GPL-3.0` variable instead.
- `ASNEEDED` is now included in the `TARGET_LDFLAGS` variable directly. The remaining definitions from `meta/conf/distro/include/as-needed.inc` have been moved to corresponding recipes.
- Support for DSA host keys has been dropped from the OpenSSH recipes. If you are still using DSA keys, you must switch over to a more secure algorithm as recommended by OpenSSH upstream.
- The `dhcp` recipe now uses the `dhcpd6.conf` configuration file in `dhcpd6.service` for IPv6 DHCP rather than re-using `dhcpd.conf`, which is now reserved for IPv4.

## 24.15. Moving to the Yocto Project 2.7 Release¶

This section provides migration information for moving to the Yocto Project 2.7 Release from the prior release.

### 24.15.1. BitBake Changes¶

The following changes have been made to BitBake:

- BitBake now checks anonymous Python functions and pure Python functions (e.g. `def funcname:`) in the metadata for tab indentation. If found, BitBake produces a warning.
- Bitbake now checks `BBFILE_COLLECTIONS` for duplicate entries and triggers an error if any are found.

### 24.15.2. Eclipse™ Support Removed¶

Support for the Eclipse IDE has been removed. Support continues for those releases prior to 2.7 that did include support. The 2.7 release does not include the Eclipse Yocto plugin.

### 24.15.3. qemu-native Splits the System and User-Mode Parts¶

The system and user-mode parts of `qemu-native` are now split. `qemu-native` provides the user-mode components and `qemu-system-native` provides the system components. If you have recipes that depend on QEMU's system emulation functionality at build time, they should now depend upon `qemu-system-native` instead of `qemu-native`.

### 24.15.4. The upstream-tracking.inc File Has Been Removed¶

The previously deprecated `upstream-tracking.inc` file is now removed. Any `UPSTREAM_TRACKING*` variables are now set in the corresponding recipes instead.

Remove any references you have to the `upstream-tracking.inc` file in your configuration.

### 24.15.5. The DISTRO\_FEATURES\_LIBC Variable Has Been Removed¶

The `DISTRO_FEATURES_LIBC` variable is no longer used. The ability to configure `glibc` using `kconfig` has been removed for quite some time making the `libc-*` features set no longer effective.

Remove any references you have to `DISTRO_FEATURES_LIBC` in your own layers.

### 24.15.6. License Value Corrections¶

The following corrections have been made to the `LICENSE` values set by recipes:

```
socat: Corrected LICENSE to be "GPLv2" rather than
 "GPLv2+".
```

```
Libgfortran: Set license to "GPL-3.0-with-GCC-exception".
```

*elfutils*: Removed "Elfutils-Exception" and set to "GPLv2" for shared libraries

### 24.15.7. Packaging Changes¶

This section provides information about packaging changes.

- *bind*: The *nsupdate* binary has been moved to the *bind-utils* package.
- *Debug split*: The default debug split has been changed to create separate source packages (i.e. *package\_name=dbg* and *package\_name=src*). If you are currently using *dbg-pkgs* in *IMAGE\_FEATURES* to bring in debug symbols and you still need the sources, you must now also add *src-pkgs* to *IMAGE\_FEATURES*. Source packages remain in the target portion of the SDK by default, unless you have set your own value for *SDKIMAGE\_FEATURES* that does not include *src-pkgs*.
- *Mount all using util-linux*: */etc/default/mountall* has moved into the *-mount* sub-package.
- *Splitting binaries using util-linux*: *util-linux* now splits each binary into its own package for fine-grained control. The main *util-linux* package pulls in the individual binary packages using the *RRECOMMENDS* and *RDEPENDS* variables. As a result, existing images should not see any changes assuming *NO\_RECOMMENDATIONS* is not set.
- *netbase/base-files*: */etc/hosts* has moved from *netbase* to *base-files*.
- *tzdata*: The main package has been converted to an empty meta package that pulls in all *tzdata* packages by default.
- *lrzsz*: This package has been removed from *packagegroup-self-hosted* and *packagegroup-core-tools-testapps*. The X/Y/ZModem support is less likely to be needed on modern systems. If you are relying on these packagegroups to include the *lrzsz* package in your image, you now need to explicitly add the package.

### 24.15.8. Removed Recipes¶

The following recipes have been removed:

*gcc*: Drop version 7.3 recipes. Version 8.3 now remains.

*linux-yocto*: Drop versions 4.14 and 4.18 recipes. Versions 4.19 and 5.0 remain.

*go*: Drop version 1.9 recipes. Versions 1.11 and 1.12 remain.

*xvideo-tests*: Became obsolete.

*libart-lgpl*: Became obsolete.

*gtk-icon-utils-native*: These tools are now provided by *gtk+3-native*

*gcc-cross-initial*: No longer needed. *gcc-cross/gcc-crosssdk* is now used instead.

*gcc-crosssdk-initial*: No longer needed. *gcc-cross/gcc-crosssdk* is now used instead.

*glibc-initial*: Removed because the benefits of having it for *site\_config* are currently outweighed by the cost of building the recipe.

### 24.15.9. Removed Classes¶

The following classes have been removed:

*distutils-tools*: This class was never used.

*bugzilla.bbclass*: Became obsolete.

*distrodata*: This functionally has been replaced by a more modern tinfoil-based implementation.

### 24.15.10. Miscellaneous Changes¶

The following miscellaneous changes occurred:

- The *distro* subdirectory of the Poky repository has been removed from the top-level *scripts* directory.
- Perl now builds for the target using *perl-cross* for better maintainability and improved build performance. This change should not present any problems unless you have heavily customized your Perl recipe.
- *arm-tunes*: Removed the "-march" option if *mcpu* is already added.
- *update-alternatives*: Convert file renames to *PACKAGE\_PREPROCESS\_FUNCS*
- *base/pixbufcache*: Obsolete *ssstatecompletions* code has been removed.
- *native* class: *RDEPENDS* handling has been enabled.
- *inetutils*: This recipe has rsh disabled.

## 24.16. Moving to the Yocto Project 3.0 Release¶

This section provides migration information for moving to the Yocto Project 3.0 Release from the prior release.

### 24.16.1. Init System Selection¶

Changing the init system manager previously required setting a number of different variables. You can now change the manager by setting the `INIT_MANAGER` variable and the corresponding include files (i.e. `conf/distro/include/init-manager-*.conf`). Include files are provided for four values: "none", "sysvinit", "systemd", and "mdev-busybox". The default value, "none", for `INIT_MANAGER` should allow your current settings to continue working. However, it is advisable to explicitly set `INIT_MANAGER`.

### 24.16.2. LSB Support Removed¶

Linux Standard Base (LSB) as a standard is not current, and is not well suited for embedded applications. Support can be continued in a separate layer if needed. However, presently LSB support has been removed from the core.

As a result of this change, the `poky-lsb` derivative distribution configuration that was also used for testing alternative configurations has been replaced with a `poky-altcfg` distribution that has LSB parts removed.

### 24.16.3. Removed Recipes¶

The following recipes have been removed.

- `core-image-lsb-dev`: Part of removed LSB support.
- `core-image-lsb`: Part of removed LSB support.
- `core-image-lsb-sdk`: Part of removed LSB support.
- `cve-check-tool`: Functionally replaced by the `cve-update-db` recipe and `cve-check` class.
- `eglinf`: No longer maintained. `eglinf` from `mesa-demos` is an adequate and maintained alternative.
- `gcc-8.3`: Version 8.3 removed. Replaced by 9.2.
- `gnome-themes-standard`: Only needed by `gtk+ 2.x`, which has been removed.
- `gtk+`: GTK+ 2 is obsolete and has been replaced by `gtk+3`.
- `irda-utils`: Has become obsolete. IrDA support has been removed from the Linux kernel in version 4.17 and later.
- `libnewt-python`: `libnewt` Python support merged into main `libnewt` recipe.
- `libsdl`: Replaced by newer `libsdl2`.
- `libx11-diet`: Became obsolete.
- `libxx86dga`: Removed obsolete client library.
- `libxx86misc`: Removed. Library is redundant.
- `linux-yocto`: Version 5.0 removed, which is now redundant (5.2 / 4.19 present).
- `lsbinitscripts`: Part of removed LSB support.
- `lsb`: Part of removed LSB support.
- `lsbtest`: Part of removed LSB support.
- `openssl10`: Replaced by newer `openssl` version 1.1.
- `packagegroup-core-lsb`: Part of removed LSB support.
- `python-nose`: Removed the Python 2.x version of the recipe.
- `python-numpy`: Removed the Python 2.x version of the recipe.
- `python-scons`: Removed the Python 2.x version of the recipe.
- `source-highlight`: No longer needed.
- `stress`: Replaced by `stress-ng`.
- `vulkan`: Split into `vulkan-loader`, `vulkan-headers`, and `vulkan-tools`.
- `weston-conf`: Functionality moved to `weston-init`.

### 24.16.4. Packaging Changes¶

The following packaging changes have occurred.

- The [Epiphany](#) browser has been dropped from `packagegroup-self-hosted` as it has not been needed inside `build-appliance-image` for quite some time and was causing resource problems.
- `libcap-ng` Python support has been moved to a separate `libcap-ng-python` recipe to streamline the build process when the Python bindings are not needed.
- `libdrm` now packages the file `amdgpu.ids` into a separate `libdrm-amdgpu` package.
- `python3`: The `runpy` module is now in the `python3-core` package as it is required to support the common "python3 -m" command usage.
- `distcc` now provides separate `distcc-client` and `distcc-server` packages as typically one or the other are needed, rather than both.
- `python*-setuptools` recipes now separately package the `pkg_resources` module in a `python-pkg-resources` / `python3-pkg-resources` package as the module is useful independent of the rest of the `setuptools` package. The main `python-setuptools` / `python3-setuptools` package depends on this new package so you should only need to update dependencies unless you want to take advantage of the increased granularity.

### 24.16.5. CVE Checking¶

`cve-check-tool` has been functionally replaced by a new `cve-update-db` recipe and functionality built into the `cve-check` class. The result uses NVD JSON data feeds rather than the deprecated XML feeds that `cve-check-tool` was using, supports CVSSv3 scoring, and makes other improvements.

Additionally, the `CVE_CHECK_CVE_WHITELIST` variable has been replaced by `CVE_CHECK_WHITELIST`.

### 24.16.6. Bitbake Changes¶

The following BitBake changes have occurred.

- `addtask` statements now properly validate dependent tasks. Previously, an invalid task was silently ignored. With this change, the invalid task generates a warning.
- Other invalid `addtask` and `deltask` usages now trigger these warnings: "multiple target tasks arguments with `addtask / deltask`", and "multiple before/after clauses".
- The "multiconfig" prefix is now shortened to "mc". "multiconfig" will continue to work, however it may be removed in a future release.
- The `bitbake -g` command no longer generates a `recipe-depends.dot` file as the contents (i.e. a reprocessed version of `task-depends.dot`) were confusing.
- The `bb.build.FuncFailed` exception, previously raised by `bb.build.exec_func()` when certain other exceptions have occurred, has been removed. The real underlying exceptions will be raised instead. If you have calls to `bb.build.exec_func()` in custom classes or `tinfoil-using` scripts, any references to `bb.build.FuncFailed` should be cleaned up.
- Additionally, the `bb.build.exec_func()` no longer accepts the "pythonexception" parameter. The function now always raises exceptions. Remove this argument in any calls to `bb.build.exec_func()` in custom classes or scripts.
- The `BB_SETSCENE_VERIFY_FUNCTION2` is no longer used. In the unlikely event that you have any references to it, they should be removed.
- The `RunQueueExecuteScenequeue` and `RunQueueExecuteTasks` events have been removed since `setscene` tasks are now executed as part of the normal runqueue. Any event handling code in custom classes or scripts that handles these two events need to be updated.
- The arguments passed to functions used with `BB_HASHCHECK_FUNCTION` have changed. If you are using your own custom hash check function, see <http://git.yoctoproject.org/cgit/cgit.cgi/poky/commit/?id=40a5e193c4ba45c928fcd899415ea56b5417725> for details.
- Task specifications in `BB_TASKDEPDATA` and class implementations used in signature generator classes now use "<fn>:" everywhere rather than the "." delimiter that was being used in some places. This change makes it consistent with all areas in the code. Custom signature generator classes and code that reads `BB_TASKDEPDATA` need to be updated to use ":" as a separator rather than ".".

### 24.16.7. Sanity Checks¶

The following sanity check changes occurred.

- `SRC_URI` is now checked for usage of two problematic items:
  - "\${PN}" prefix/suffix use - Warnings always appear if `${PN}` is used. You must fix the issue regardless of whether `multiconfig` or anything else that would cause prefixing/suffixing to happen.
  - Github archive tarballs - these are not guaranteed to be stable. Consequently, it is likely that the tarballs will be refreshed and thus the `SRC_URI` checksums will fail to apply. It is recommended that you fetch either an official release tarball or a specific revision from the actual Git repository instead.

Either one of these items now trigger a warning by default. If you wish to disable this check, remove `src-uri-bad` from `WARN_OA`.

- The `file-rdeps` runtime dependency check no longer expands `RDEPENDS` recursively as there is no mechanism to ensure they can be fully computed, and thus races sometimes result in errors either showing up or not. Thus, you might now see errors for missing runtime dependencies that were previously satisfied recursively. Here is an example: package A contains a shell script starting with `#!/bin/bash` but has no dependency on `bash`. However, package A depends on package B, which does depend on `bash`. You need to add the missing dependency or dependencies to resolve the warning.
- Setting `DEPENDS_${PN}` anywhere (i.e. typically in a recipe) now triggers an error. The error is triggered because `DEPENDS` is not a package-specific variable unlike `RDEPENDS`. You should set `DEPENDS` instead.
- `systemd` currently does not work well with the `musl` C library because only upstream officially supports linking the library with `glibc`. Thus, a warning is shown when building `systemd` in conjunction with `musl`.

### 24.16.8. Miscellaneous Changes¶

The following miscellaneous changes have occurred.

- The `gnome` class has been removed because it now does very little. You should update recipes that previously inherited this class to do the following:
 

```
inherit gnomebase gtk-icon-cache gconf mime
```
- The `meta/recipes-kernel/linux/linux-dtb.inc` file has been removed. This file was previously deprecated in favor of setting `KERNEL_DEVICETREE` in any kernel recipe and only produced a warning. Remove any `include` or `require` statements pointing to this file.

- `TARGET_CFLAGS`, `TARGET_CPPFLAGS`, `TARGET_CXXFLAGS`, and `TARGET_LDFLAGS` are no longer exported to the external environment. This change did not require any changes to core recipes, which is a good indicator that no changes will be required. However, if for some reason the software being built by one of your recipes is expecting these variables to be set, then building the recipe will fail. In such cases, you must either export the variable or variables in the recipe or change the scripts so that exporting is not necessary.
- You must change the host distro identifier used in `NATIVELSBSTRING` to use all lowercase characters even if it does not contain a version number. This change is necessary only if you are not using `uninative` and `SANITY_TESTED_DISTROS`.
- In the `base-files` recipe, writing the hostname into `/etc/hosts` and `/etc/hostname` is now done within the main `do_install` function rather than in the `do_install_basefilesissue` function. The reason for the change is because `do_install_basefilesissue` is more easily overridden without having to duplicate the hostname functionality. If you have done the latter (e.g. in a `base-files` bbappend), then you should remove it from your customized `do_install_basefilesissue` function.
- The `wic --expand` command now uses commas to separate "key:value" pairs rather than hyphens.

## Note

The `wic` command-line help is not updated.

You must update any scripts or commands where you use `wic --expand` with multiple "key:value" pairs.

- UEFI image variable settings have been moved from various places to a central `conf/image-uefi.conf`. This change should not influence any existing configuration as the `meta/conf/image-uefi.conf` in the core metadata sets defaults that can be overridden in the same manner as before.
- `conf/distro/include/world-broken.inc` has been removed. For cases where certain recipes need to be disabled when using the `musl` C library, these recipes now have `COMPATIBLE_HOST_libc-musl` set with a comment that explains why.

## Chapter 25. Source Directory Structure¶

### 25.1. Top-Level Core Components

[25.1.1. bitbake/](#)  
[25.1.2. build/](#)  
[25.1.3. documentation/](#)  
[25.1.4. meta/](#)  
[25.1.5. meta-poky/](#)  
[25.1.6. meta-yocto-bsp/](#)  
[25.1.7. meta-selftest/](#)  
[25.1.8. meta-skeleton/](#)  
[25.1.9. scripts/](#)  
[25.1.10. oe-init-build-env](#)  
[25.1.11. LICENSE, README, and README.hardware](#)

### 25.2. The Build Directory - `build/`

[25.2.1. build/buildhistory](#)  
[25.2.2. build/conf/local.conf](#)  
[25.2.3. build/conf/bblayers.conf](#)  
[25.2.4. build/conf/sanity\\_info](#)  
[25.2.5. build/downloads/](#)  
[25.2.6. build/sstate-cache/](#)  
[25.2.7. build/tmp/](#)  
[25.2.8. build/tmp/buildstats/](#)  
[25.2.9. build/tmp/cache/](#)  
[25.2.10. build/tmp/deploy/](#)  
[25.2.11. build/tmp/deploy/deb/](#)  
[25.2.12. build/tmp/deploy/rpm/](#)  
[25.2.13. build/tmp/deploy/ipk/](#)  
[25.2.14. build/tmp/deploy/licenses/](#)  
[25.2.15. build/tmp/deploy/images/](#)  
[25.2.16. build/tmp/deploy/sdk/](#)  
[25.2.17. build/tmp/sstate-control/](#)  
[25.2.18. build/tmp/sysroots-components/](#)  
[25.2.19. build/tmp/sysroots/](#)  
[25.2.20. build/tmp/stamps/](#)  
[25.2.21. build/tmp/log/](#)  
[25.2.22. build/tmp/work/](#)  
[25.2.23. build/tmp/work/tunearch/recipeName/version/](#)  
[25.2.24. build/tmp/work-shared/](#)

### 25.3. The Metadata - `meta/`

[25.3.1. meta/classes/](#)  
[25.3.2. meta/conf/](#)  
[25.3.3. meta/conf/machine/](#)



[25.3.4.meta/conf/distro/](#)  
[25.3.5.meta/conf/machine-sdk/](#)  
[25.3.6.meta/files/](#)  
[25.3.7.meta/lib/](#)  
[25.3.8.meta/recipes-bsp/](#)  
[25.3.9.meta/recipes-connectivity/](#)  
[25.3.10.meta/recipes-core/](#)  
[25.3.11.meta/recipes-devtools/](#)  
[25.3.12.meta/recipes-extended/](#)  
[25.3.13.meta/recipes-gnome/](#)  
[25.3.14.meta/recipes-graphics/](#)  
[25.3.15.meta/recipes-kernel/](#)  
[25.3.16.meta/recipes-lsb4/](#)  
[25.3.17.meta/recipes-multimedia/](#)  
[25.3.18.meta/recipes-rt/](#)  
[25.3.19.meta/recipes-sato/](#)  
[25.3.20.meta/recipes-support/](#)  
[25.3.21.meta/site/](#)  
[25.3.22.meta/recipes.txt](#)

The [Source Directory](#) consists of several components. Understanding them and knowing where they are located is key to using the Yocto Project well. This chapter describes the Source Directory and gives information about the various files and directories.

For information on how to establish a local Source Directory on your development system, see the "[Locating Yocto Project Source Files](#)" section in the Yocto Project Development Tasks Manual.

## Note

The OpenEmbedded build system does not support file or directory names that contain spaces. Be sure that the Source Directory you use does not contain these types of names.

## 25.1. Top-Level Core Components¶

This section describes the top-level components of the [Source Directory](#).

### 25.1.1. bitbake/¶

This directory includes a copy of BitBake for ease of use. The copy usually matches the current stable BitBake release from the BitBake project. BitBake, a [Metadata](#) interpreter, reads the Yocto Project Metadata and runs the tasks defined by that data. Failures are usually from the Metadata and not from BitBake itself. Consequently, most users do not need to worry about BitBake.

When you run the `bitbake` command, the main BitBake executable, which resides in the `bitbake/bin/` directory, starts. Sourcing the environment setup script (i.e. `oe-init-build-env`) places the `scripts` and `bitbake/bin` directories (in that order) into the shell's `PATH` environment variable.

For more information on BitBake, see the BitBake User Manual.

### 25.1.2. build/¶

This directory contains user configuration files and the output generated by the OpenEmbedded build system in its standard configuration where the source tree is combined with the output. The [Build Directory](#) is created initially when you `source` the OpenEmbedded build environment setup script (i.e. `oe-init-build-env`).

It is also possible to place output and configuration files in a directory separate from the [Source Directory](#) by providing a directory name when you `source` the setup script. For information on separating output from your local Source Directory files, see the "[oe-init-build-env](#)" section.

### 25.1.3. documentation/¶

This directory holds the source for the Yocto Project documentation as well as templates and tools that allow you to generate PDF and HTML versions of the manuals. Each manual is contained in a sub-folder. For example, the files for this manual reside in the `ref-manual/` directory.

### 25.1.4. meta/¶

This directory contains the OpenEmbedded-Core metadata. The directory holds recipes, common classes, and machine configuration for emulated targets (`qemux86`, `qemuarm`, and so forth.)

### 25.1.5. meta-poky/¶

This directory contains the configuration for the Poky reference distribution.

### 25.1.6. meta-yocto-bsp/¶

This directory contains the Yocto Project reference hardware Board Support Packages (BSPs). For more information on BSPs, see the Yocto Project Board Support Package (BSP) Developer's Guide.



### 25.1.7. meta-selftest/¶

This directory adds additional recipes and append files used by the OpenEmbedded selftests to verify the behavior of the build system.

You do not have to add this layer to your `bblayers.conf` file unless you want to run the selftests.

### 25.1.8. meta-skeleton/¶

This directory contains template recipes for BSP and kernel development.

### 25.1.9. scripts/¶

This directory contains various integration scripts that implement extra functionality in the Yocto Project environment (e.g. QEMU scripts). The `oe-init-build-env` script appends this directory to the shell's `PATH` environment variable.

The `scripts` directory has useful scripts that assist in contributing back to the Yocto Project, such as `create-pull-request` and `send-pull-request`.

### 25.1.10. oe-init-build-env¶

This script sets up the OpenEmbedded build environment. Running this script with the `SOURCE` command in a shell makes changes to `PATH` and sets other core BitBake variables based on the current working directory. You need to run an environment setup script before running BitBake commands. The script uses other scripts within the `scripts` directory to do the bulk of the work.

When you run this script, your Yocto Project environment is set up, a Build Directory is created, your working directory becomes the Build Directory, and you are presented with a list of common BitBake targets. Here is an example:

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

Shell environment set up for builds.

You can now run 'bitbake <target>'

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

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

The script gets its default list of common targets from the `conf-notes.txt` file, which is found in the `meta-poky` directory within the Source Directory. Should you have custom distributions, it is very easy to modify this configuration file to include your targets for your distribution. See the "Creating a Custom Template Configuration Directory" section in the Yocto Project Development Tasks Manual for more information.

By default, running this script without a Build Directory argument creates the `build` directory in your current working directory. If you provide a Build Directory argument when you `SOURCE` the script, you direct the OpenEmbedded build system to create a Build Directory of your choice. For example, the following command creates a Build Directory named `mybuilds` that is outside of the Source Directory:

```
$ source oe-init-build-env ~/mybuilds
```

The OpenEmbedded build system uses the template configuration files, which are found by default in the `meta-poky/conf` directory in the Source Directory. See the "Creating a Custom Template Configuration Directory" section in the Yocto Project Development Tasks Manual for more information.

## Note

The OpenEmbedded build system does not support file or directory names that contain spaces. If you attempt to run the `oe-init-build-env` script from a Source Directory that contains spaces in either the filenames or directory names, the script returns an error indicating no such file or directory. Be sure to use a Source Directory free of names containing spaces.

### 25.1.11. LICENSE, README, and README.hardware¶

These files are standard top-level files.

## 25.2. The Build Directory - build/¶

The OpenEmbedded build system creates the Build Directory when you run the build environment setup scripts (i.e. `oe-init-build-env`).

If you do not give the Build Directory a specific name when you run a setup script, the name defaults to `build`.

The `TOPDIR` variable points to the Build Directory.

### 25.2.1. build/buildhistory¶

The OpenEmbedded build system creates this directory when you enable the build history feature. The directory tracks build information into image, packages, and SDK subdirectories. For information on the build history feature, see the "Maintaining Build Output Quality"

section in the Yocto Project Development Tasks Manual.

### 25.2.2. build/conf/local.conf

This configuration file contains all the local user configurations for your build environment. The `local.conf` file contains documentation on the various configuration options. Any variable set here overrides any variable set elsewhere within the environment unless that variable is hard-coded within a file (e.g. by using `'='` instead of `'?='`). Some variables are hard-coded for various reasons but these variables are relatively rare.

Edit this file to set the `MACHINE` for which you want to build, which package types you wish to use (`PACKAGE_CLASSES`), and the location from which you want to access downloaded files (`DL_DIR`).

If `local.conf` is not present when you start the build, the OpenEmbedded build system creates it from `local.conf.sample` when you `source` the top-level build environment setup script (i.e. `oe-init-build-env`).

The source `local.conf.sample` file used depends on the `$TEMPLATECONF` script variable, which defaults to `meta-poky/conf` when you are building from the Yocto Project development environment and defaults to `meta/conf` when you are building from the OpenEmbedded-Core environment. Because the script variable points to the source of the `local.conf.sample` file, this implies that you can configure your build environment from any layer by setting the variable in the top-level build environment setup script as follows:

```
TEMPLATECONF=your_layer/conf
```

Once the build process gets the sample file, it uses `sed` to substitute final `${OEROOT}` values for all `##OEROOT##` values.

#### Note

You can see how the `TEMPLATECONF` variable is used by looking at the `scripts/oe-setup-buildidir` script in the [Source Directory](#). You can find the Yocto Project version of the `local.conf.sample` file in the `meta-poky/conf` directory.

### 25.2.3. build/conf/bblayers.conf

This configuration file defines `layers`, which are directory trees, traversed (or walked) by BitBake. The `bblayers.conf` file uses the `BBLAYERS` variable to list the layers BitBake tries to find.

If `bblayers.conf` is not present when you start the build, the OpenEmbedded build system creates it from `bblayers.conf.sample` when you `source` the top-level build environment setup script (i.e. `oe-init-build-env`).

The source `bblayers.conf.sample` file used depends on the `$TEMPLATECONF` script variable, which defaults to `meta-poky/conf` when you are building from the Yocto Project development environment and defaults to `meta/conf` when you are building from the OpenEmbedded-Core environment. Because the script variable points to the source of the `bblayers.conf.sample` file, this implies that you can base your build from any layer by setting the variable in the top-level build environment setup script as follows:

```
TEMPLATECONF=your_layer/conf
```

Once the build process gets the sample file, it uses `sed` to substitute final `${OEROOT}` values for all `##OEROOT##` values.

#### Note

You can see how the `TEMPLATECONF` variable `scripts/oe-setup-buildidir` script in the [Source Directory](#). You can find the Yocto Project version of the `bblayers.conf.sample` file in the `meta-poky/conf` directory.

### 25.2.4. build/conf/sanity\_info

This file indicates the state of the sanity checks and is created during the build.

### 25.2.5. build/downloads/

This directory contains downloaded upstream source tarballs. You can reuse the directory for multiple builds or move the directory to another location. You can control the location of this directory through the `DL_DIR` variable.

### 25.2.6. build/sstate-cache/

This directory contains the shared state cache. You can reuse the directory for multiple builds or move the directory to another location. You can control the location of this directory through the `SSTATE_DIR` variable.

### 25.2.7. build/tmp/

The OpenEmbedded build system creates and uses this directory for all the build system's output. The `TMPDIR` variable points to this directory.

BitBake creates this directory if it does not exist. As a last resort, to clean up a build and start it from scratch (other than the downloads), you can remove everything in the `tmp` directory or get rid of the directory completely. If you do, you should also

completely remove the `build/sstate-cache` directory.

### 25.2.8. `build/tmp/buildstats/`

This directory stores the build statistics.

### 25.2.9. `build/tmp/cache/`

When BitBake parses the metadata (recipes and configuration files), it caches the results in `build/tmp/cache/` to speed up future builds. The results are stored on a per-machine basis.

During subsequent builds, BitBake checks each recipe (together with, for example, any files included or appended to it) to see if they have been modified. Changes can be detected, for example, through file modification time (mtime) changes and hashing of file contents. If no changes to the file are detected, then the parsed result stored in the cache is reused. If the file has changed, it is reparsed.

### 25.2.10. `build/tmp/deploy/`

This directory contains any "end result" output from the OpenEmbedded build process. The `DEPLOY_DIR` variable points to this directory. For more detail on the contents of the `deploy` directory, see the "[Images](#)" and "[Application Development SDK](#)" sections in the Yocto Project Overview and Concepts Manual.

### 25.2.11. `build/tmp/deploy/deb/`

This directory receives any `.deb` packages produced by the build process. The packages are sorted into feeds for different architecture types.

### 25.2.12. `build/tmp/deploy/rpm/`

This directory receives any `.rpm` packages produced by the build process. The packages are sorted into feeds for different architecture types.

### 25.2.13. `build/tmp/deploy/ipk/`

This directory receives `.ipk` packages produced by the build process.

### 25.2.14. `build/tmp/deploy/licenses/`

This directory receives package licensing information. For example, the directory contains sub-directories for `bash`, `busybox`, and `glibc` (among others) that in turn contain appropriate `COPYING` license files with other licensing information. For information on licensing, see the "[Maintaining Open Source License Compliance During Your Product's Lifecycle](#)" section in the Yocto Project Development Tasks Manual.

### 25.2.15. `build/tmp/deploy/images/`

This directory receives complete filesystem images. If you want to flash the resulting image from a build onto a device, look here for the image.

Be careful when deleting files in this directory. You can safely delete old images from this directory (e.g. `core-image-*`). However, the kernel (`*zImage*`, `*uImage*`, etc.), bootloader and other supplementary files might be deployed here prior to building an image. Because these files are not directly produced from the image, if you delete them they will not be automatically re-created when you build the image again.

If you do accidentally delete files here, you will need to force them to be re-created. In order to do that, you will need to know the target that produced them. For example, these commands rebuild and re-create the kernel files:

```
$ bitbake -c clean virtual/kernel
$ bitbake virtual/kernel
```

### 25.2.16. `build/tmp/deploy/sdk/`

The OpenEmbedded build system creates this directory to hold toolchain installer scripts, which when executed, install the sysroot that matches your target hardware. You can find out more about these installers in the "[Building an SDK Installer](#)" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

### 25.2.17. `build/tmp/sstate-control/`

The OpenEmbedded build system uses this directory for the shared state manifest files. The shared state code uses these files to record the files installed by each sstate task so that the files can be removed when cleaning the recipe or when a newer version is about to be installed. The build system also uses the manifests to detect and produce a warning when files from one task are overwriting those from another.

### 25.2.18. `build/tmp/sysroots-components/`

This directory is the location of the sysroot contents that the task `do_prepare_recipe_sysroot` links or copies into the recipe-specific sysroot for each recipe listed in `DEPENDS`. Population of this directory is handled through shared state, while the path is specified by the `COMPONENTS_DIR` variable. Apart from a few unusual circumstances, handling of the `sysroots-components` directory should be automatic, and recipes should not directly reference `build/tmp/sysroots-components`.

### 25.2.19. build/tmp/sysroots/¶

Previous versions of the OpenEmbedded build system used to create a global shared sysroot per machine along with a native sysroot. Beginning with the 3.0.1 version of the Yocto Project, sysroots exist in recipe-specific [WORKDIR](#) directories. Thus, the `build/tmp/sysroots/` directory is unused.

#### Note

The `build/tmp/sysroots/` directory can still be populated using the `bitbake build-sysroots` command and can be used for compatibility in some cases. However, in general it is not recommended to populate this directory. Individual recipe-specific sysroots should be used.

### 25.2.20. build/tmp/stamps/¶

This directory holds information that BitBake uses for accounting purposes to track what tasks have run and when they have run. The directory is sub-divided by architecture, package name, and version. Following is an example:

```
stamps/all-poky-linux/distcc-config/1.0-r0.do_build-2fdd...2do
```

Although the files in the directory are empty of data, BitBake uses the filenames and timestamps for tracking purposes.

For information on how BitBake uses stamp files to determine if a task should be rerun, see the "[Stamp Files and the Rerunning of Tasks](#)" section in the Yocto Project Overview and Concepts Manual.

### 25.2.21. build/tmp/log/¶

This directory contains general logs that are not otherwise placed using the package's [WORKDIR](#). Examples of logs are the output from the `do_check_pkg` or `do_distro_check` tasks. Running a build does not necessarily mean this directory is created.

### 25.2.22. build/tmp/work/¶

This directory contains architecture-specific work sub-directories for packages built by BitBake. All tasks execute from the appropriate work directory. For example, the source for a particular package is unpacked, patched, configured and compiled all within its own work directory. Within the work directory, organization is based on the package group and version for which the source is being compiled as defined by the [WORKDIR](#).

It is worth considering the structure of a typical work directory. As an example, consider `linux-yocto-kernel-3.0` on the machine `qemux86` built within the Yocto Project. For this package, a work directory of `tmp/work/qemux86-poky-linux/linux-yocto/3.0+git1+<...>`, referred to as the [WORKDIR](#), is created. Within this directory, the source is unpacked to `linux-qemux86-standard-build` and then patched by Quilt. (See the "[Using Quilt in Your Workflow](#)" section in the Yocto Project Development Tasks Manual for more information.) Within the `linux-qemux86-standard-build` directory, standard Quilt directories `linux-3.0/patches` and `linux-3.0/.pc` are created, and standard Quilt commands can be used.

There are other directories generated within [WORKDIR](#). The most important directory is `WORKDIR/temp/`, which has log files for each task (`log.do_*.pid`) and contains the scripts BitBake runs for each task (`run.do_*.pid`). The `WORKDIR/image/` directory is where "make install" places its output that is then split into sub-packages within `WORKDIR/packages-split/`.

### 25.2.23. build/tmp/work/tunearch/recipeName/version/¶

The recipe work directory - `${WORKDIR}`.

As described earlier in the "[build/tmp/sysroots/](#)" section, beginning with the 3.0.1 release of the Yocto Project, the OpenEmbedded build system builds each recipe in its own work directory (i.e. [WORKDIR](#)). The path to the work directory is constructed using the architecture of the given build (e.g. [TUNE\\_PKGARCH](#), [MACHINE\\_ARCH](#), or "allarch"), the recipe name, and the version of the recipe (i.e. [PE:PV-PR](#)).

A number of key subdirectories exist within each recipe work directory:

- `${WORKDIR}/temp`: Contains the log files of each task executed for this recipe, the "run" files for each executed task, which contain the code run, and a `log.task_order` file, which lists the order in which tasks were executed.
- `${WORKDIR}/image`: Contains the output of the `do_install` task, which corresponds to the `${D}` variable in that task.
- `${WORKDIR}/pseudo`: Contains the pseudo database and log for any tasks executed under pseudo for the recipe.
- `${WORKDIR}/sysroot-destdir`: Contains the output of the `do_populate_sysroot` task.
- `${WORKDIR}/package`: Contains the output of the `do_package` task before the output is split into individual packages.
- `${WORKDIR}/packages-split`: Contains the output of the `do_package` task after the output has been split into individual packages. Subdirectories exist for each individual package created by the recipe.
- `${WORKDIR}/recipe-sysroot`: A directory populated with the target dependencies of the recipe. This directory looks like the target filesystem and contains libraries that the recipe might need to link against (e.g. the C library).
- `${WORKDIR}/recipe-sysroot-native`: A directory populated with the native dependencies of the recipe. This directory contains the tools the recipe needs to build (e.g. the compiler, Autoconf, libtool, and so forth).
- `${WORKDIR}/build`: This subdirectory applies only to recipes that support builds where the source is separate from the build artifacts. The OpenEmbedded build system uses this directory as a separate build directory (i.e. `${B}`).

### 25.2.24. build/tmp/work-shared/¶

For efficiency, the OpenEmbedded build system creates and uses this directory to hold recipes that share a work directory with other recipes. In practice, this is only used for `gcc` and its variants (e.g. `gcc-cross`, `libgcc`, `gcc-runtime`, and so forth).

## 25.3. The Metadata - meta/¶

As mentioned previously, [Metadata](#) is the core of the Yocto Project. Metadata has several important subdivisions:

### 25.3.1. meta/classes/¶

This directory contains the `*.bbclass` files. Class files are used to abstract common code so it can be reused by multiple packages. Every package inherits the `base.bbclass` file. Examples of other important classes are `autotools.bbclass`, which in theory allows any Autotool-enabled package to work with the Yocto Project with minimal effort. Another example is `kernel.bbclass` that contains common code and functions for working with the Linux kernel. Functions like image generation or packaging also have their specific class files such as `image.bbclass`, `rootfs_*.bbclass` and `package*.bbclass`.

For reference information on classes, see the "[Classes](#)" chapter.

### 25.3.2. meta/conf/¶

This directory contains the core set of configuration files that start from `bitbake.conf` and from which all other configuration files are included. See the include statements at the end of the `bitbake.conf` file and you will note that even `local.conf` is loaded from there. While `bitbake.conf` sets up the defaults, you can often override these by using the `(local.conf)` file, machine file or the distribution configuration file.

### 25.3.3. meta/conf/machine/¶

This directory contains all the machine configuration files. If you set `MACHINE = "qemux86"`, the OpenEmbedded build system looks for a `qemux86.conf` file in this directory. The `include` directory contains various data common to multiple machines. If you want to add support for a new machine to the Yocto Project, look in this directory.

### 25.3.4. meta/conf/distro/¶

The contents of this directory controls any distribution-specific configurations. For the Yocto Project, the `defaultsetup.conf` is the main file here. This directory includes the versions and the `SRCDATE` definitions for applications that are configured here. An example of an alternative configuration might be `poky-bleeding.conf`. Although this file mainly inherits its configuration from Poky.

### 25.3.5. meta/conf/machine-sdk/¶

The OpenEmbedded build system searches this directory for configuration files that correspond to the value of `SDKMACHINE`. By default, 32-bit and 64-bit x86 files ship with the Yocto Project that support some SDK hosts. However, it is possible to extend that support to other SDK hosts by adding additional configuration files in this subdirectory within another layer.

### 25.3.6. meta/files/¶

This directory contains common license files and several text files used by the build system. The text files contain minimal device information and lists of files and directories with known permissions.

### 25.3.7. meta/lib/¶

This directory contains OpenEmbedded Python library code used during the build process.

### 25.3.8. meta/recipes-bsp/¶

This directory contains anything linking to specific hardware or hardware configuration information such as "u-boot" and "grub".

### 25.3.9. meta/recipes-connectivity/¶

This directory contains libraries and applications related to communication with other devices.

### 25.3.10. meta/recipes-core/¶

This directory contains what is needed to build a basic working Linux image including commonly used dependencies.

### 25.3.11. meta/recipes-devtools/¶

This directory contains tools that are primarily used by the build system. The tools, however, can also be used on targets.

### 25.3.12. meta/recipes-extended/¶

This directory contains non-essential applications that add features compared to the alternatives in core. You might need this directory for full tool functionality or for Linux Standard Base (LSB) compliance.

**25.3.13. meta/recipes-gnome/¶**

This directory contains all things related to the GTK+ application framework.

**25.3.14. meta/recipes-graphics/¶**

This directory contains X and other graphically related system libraries

**25.3.15. meta/recipes-kernel/¶**

This directory contains the kernel and generic applications and libraries that have strong kernel dependencies.

**25.3.16. meta/recipes-lsb4/¶**

This directory contains recipes specifically added to support the Linux Standard Base (LSB) version 4.x.

**25.3.17. meta/recipes-multimedia/¶**

This directory contains codecs and support utilities for audio, images and video.

**25.3.18. meta/recipes-rt/¶**

This directory contains package and image recipes for using and testing the PREEMPT\_RT kernel.

**25.3.19. meta/recipes-sato/¶**

This directory contains the Sato demo/reference UI/UX and its associated applications and configuration data.

**25.3.20. meta/recipes-support/¶**

This directory contains recipes used by other recipes, but that are not directly included in images (i.e. dependencies of other recipes).

**25.3.21. meta/site/¶**

This directory contains a list of cached results for various architectures. Because certain "autoconf" test results cannot be determined when cross-compiling due to the tests not able to run on a live system, the information in this directory is passed to "autoconf" for the various architectures.

**25.3.22. meta/recipes.txt¶**

This file is a description of the contents of recipes-.\*.

**Chapter 26. Classes¶**

[26.1. allarch.bbclass](#)

[26.2. archiver.bbclass](#)

[26.3. autotools\\*.bbclass](#)

[26.4. base.bbclass](#)

[26.5. bash-completion.bbclass](#)

[26.6. bin\\_package.bbclass](#)

[26.7. binconfig.bbclass](#)

[26.8. binconfig-disabled.bbclass](#)

[26.9. blacklist.bbclass](#)

[26.10. buildhistory.bbclass](#)

[26.11. buildstats.bbclass](#)

[26.12. buildstats-summary.bbclass](#)

[26.13. ccache.bbclass](#)

[26.14. chrpath.bbclass](#)

[26.15. clutter.bbclass](#)

[26.16. cmake.bbclass](#)

[26.17. cml1.bbclass](#)

[26.18. compress\\_doc.bbclass](#)

[26.19. copyleft\\_compliance.bbclass](#)

[26.20. copyleft\\_filter.bbclass](#)

[26.21. core-image.bbclass](#)

[26.22. cpan\\*.bbclass](#)

[26.23. cross.bbclass](#)

[26.24. cross-canadian.bbclass](#)

[26.25. crosssdk.bbclass](#)

[26.26. debian.bbclass](#)

[26.27. deploy.bbclass](#)

[26.28. devshell.bbclass](#)

[26.29. devupstream.bbclass](#)

[26.30.distro\\_features\\_check.bbclass](#)  
[26.31.distutils\\*.bbclass](#)  
[26.32.distutils3\\*.bbclass](#)  
[26.33.externalsrc.bbclass](#)  
[26.34.extrausers.bbclass](#)  
[26.35.fontcache.bbclass](#)  
[26.36.fs-uuid.bbclass](#)  
[26.37.gconf.bbclass](#)  
[26.38.gettext.bbclass](#)  
[26.39.gnomebase.bbclass](#)  
[26.40.gobject-introspection.bbclass](#)  
[26.41.grub-efi.bbclass](#)  
[26.42.gsettings.bbclass](#)  
[26.43.gtk-doc.bbclass](#)  
[26.44.gtk-icon-cache.bbclass](#)  
[26.45.gtk-immodules-cache.bbclass](#)  
[26.46.gzipnative.bbclass](#)  
[26.47.icecc.bbclass](#)  
[26.48.image.bbclass](#)  
[26.49.image-buildinfo.bbclass](#)  
[26.50.image\\_types.bbclass](#)  
[26.51.image-live.bbclass](#)  
[26.52.image-mklibs.bbclass](#)  
[26.53.image-prelink.bbclass](#)  
[26.54.insane.bbclass](#)  
[26.55.insserv.bbclass](#)  
[26.56.kernel.bbclass](#)  
[26.57.kernel-arch.bbclass](#)  
[26.58.kernel-devicetree.bbclass](#)  
[26.59.kernel-fitimage.bbclass](#)  
[26.60.kernel-grub.bbclass](#)  
[26.61.kernel-module-split.bbclass](#)  
[26.62.kernel-uboot.bbclass](#)  
[26.63.kernel-uimage.bbclass](#)  
[26.64.kernel-yocto.bbclass](#)  
[26.65.kernelsrc.bbclass](#)  
[26.66.lib\\_package.bbclass](#)  
[26.67.libc\\*.bbclass](#)  
[26.68.license.bbclass](#)  
[26.69.linux-kernel-base.bbclass](#)  
[26.70.linuxloader.bbclass](#)  
[26.71.logging.bbclass](#)  
[26.72.meta.bbclass](#)  
[26.73.metadata\\_scm.bbclass](#)  
[26.74.migrate\\_localcount.bbclass](#)  
[26.75.mime.bbclass](#)  
[26.76.mirrors.bbclass](#)  
[26.77.module.bbclass](#)  
[26.78.module-base.bbclass](#)  
[26.79.multilib\\*.bbclass](#)  
[26.80.native.bbclass](#)  
[26.81.nativesdk.bbclass](#)  
[26.82.nopackages.bbclass](#)  
[26.83.npm.bbclass](#)  
[26.84.oelint.bbclass](#)  
[26.85.own-mirrors.bbclass](#)  
[26.86.package.bbclass](#)  
[26.87.package\\_deb.bbclass](#)  
[26.88.package\\_ipk.bbclass](#)  
[26.89.package\\_rpm.bbclass](#)  
[26.90.package\\_tar.bbclass](#)  
[26.91.packagedata.bbclass](#)  
[26.92.packagegroup.bbclass](#)  
[26.93.patch.bbclass](#)  
[26.94.perlnative.bbclass](#)  
[26.95.pixbufcache.bbclass](#)  
[26.96.pkgconfig.bbclass](#)  
[26.97.populate\\_sdk.bbclass](#)  
[26.98.populate\\_sdk \\*.bbclass](#)  
[26.99.prexport.bbclass](#)  
[26.100.primport.bbclass](#)  
[26.101.prserv.bbclass](#)  
[26.102.ptest.bbclass](#)  
[26.103.ptest-gnome.bbclass](#)

[26.104.python-dir.bbclass](#)  
[26.105.python3native.bbclass](#)  
[26.106.pythonnative.bbclass](#)  
[26.107.qemu.bbclass](#)  
[26.108.recipe\\_sanity.bbclass](#)  
[26.109.relocatable.bbclass](#)  
[26.110.remove-libtool.bbclass](#)  
[26.111.report-error.bbclass](#)  
[26.112.rm\\_work.bbclass](#)  
[26.113.rootfs\\*.bbclass](#)  
[26.114.sanity.bbclass](#)  
[26.115.scons.bbclass](#)  
[26.116.sdl.bbclass](#)  
[26.117.setuptools.bbclass](#)  
[26.118.setuptools3.bbclass](#)  
[26.119.sign\\_rpm.bbclass](#)  
[26.120.sip.bbclass](#)  
[26.121.siteconfig.bbclass](#)  
[26.122.siteinfo.bbclass](#)  
[26.123.spdx.bbclass](#)  
[26.124.sstate.bbclass](#)  
[26.125.staging.bbclass](#)  
[26.126.syslinux.bbclass](#)  
[26.127.systemd.bbclass](#)  
[26.128.systemd-boot.bbclass](#)  
[26.129.terminal.bbclass](#)  
[26.130.testimage\\*.bbclass](#)  
[26.131.testsdk.bbclass](#)  
[26.132.texinfo.bbclass](#)  
[26.133.tinderclient.bbclass](#)  
[26.134.toaster.bbclass](#)  
[26.135.toolchain-scripts.bbclass](#)  
[26.136.typecheck.bbclass](#)  
[26.137.uboot-config.bbclass](#)  
[26.138.uninative.bbclass](#)  
[26.139.update-alternatives.bbclass](#)  
[26.140.update-rc.d.bbclass](#)  
[26.141.useradd\\*.bbclass](#)  
[26.142.utility-tasks.bbclass](#)  
[26.143.utils.bbclass](#)  
[26.144.vala.bbclass](#)  
[26.145.waf.bbclass](#)

Class files are used to abstract common functionality and share it amongst multiple recipe (`.bb`) files. To use a class file, you simply make sure the recipe inherits the class. In most cases, when a recipe inherits a class it is enough to enable its features. There are cases, however, where in the recipe you might need to set variables or override some default behavior.

Any [Metadata](#) usually found in a recipe can also be placed in a class file. Class files are identified by the extension `.bbclass` and are usually placed in a `classes/` directory beneath the `meta*/` directory found in the [Source Directory](#). Class files can also be pointed to by [BUILDDIR](#) (e.g. `build/`) in the same way as `.conf` files in the `conf` directory. Class files are searched for in [BBPATH](#) using the same method by which `.conf` files are searched.

This chapter discusses only the most useful and important classes. Other classes do exist within the `meta/classes` directory in the Source Directory. You can reference the `.bbclass` files directly for more information.

## 26.1. `allarch.bbclass`

The `allarch` class is inherited by recipes that do not produce architecture-specific output. The class disables functionality that is normally needed for recipes that produce executable binaries (such as building the cross-compiler and a C library as pre-requisites, and splitting out of debug symbols during packaging).

### Note

Unlike some distro recipes (e.g. Debian), OpenEmbedded recipes that produce packages that depend on tunings through use of the [RDEPENDS](#) and [TUNE\\_PKGARCH](#) variables, should never be configured for all architectures using `allarch`. This is the case even if the recipes do not produce architecture-specific output.

Configuring such recipes for all architectures causes the `do_package_write_*` tasks to have different signatures for the machines with different tunings. Additionally, unnecessary rebuilds occur every time an image for a different `MACHINE` is built even when the recipe never changes.

By default, all recipes inherit the `base` and `package` classes, which enable functionality needed for recipes that produce executable output. If your recipe, for example, only produces packages that contain configuration files, media files, or scripts (e.g. Python and Perl),



then it should inherit the `allarch` class.

## 26.2. `archiver.bbclass`

The `archiver` class supports releasing source code and other materials with the binaries.

For more details on the source archiver, see the "Maintaining Open Source License Compliance During Your Product's Lifecycle" section in the Yocto Project Development Tasks Manual. You can also see the `ARCHIVER_MODE` variable for information about the variable flags (varflags) that help control archive creation.

## 26.3. `autotools*.bbclass`

The `autotools*` classes support Autotooled packages.

The `autoconf`, `automake`, and `libtool` packages bring standardization. This class defines a set of tasks (e.g. `configure`, `compile` and so forth) that work for all Autotooled packages. It should usually be enough to define a few standard variables and then simply `inherit autotools`. These classes can also work with software that emulates Autotools. For more information, see the "Autotooled Package" section in the Yocto Project Development Tasks Manual.

By default, the `autotools*` classes use out-of-tree builds (i.e. `autotools.bbclass` building with `B != S`).

If the software being built by a recipe does not support using out-of-tree builds, you should have the recipe inherit the `autotools-brokensep` class. The `autotools-brokensep` class behaves the same as the `autotools` class but builds with `B == S`. This method is useful when out-of-tree build support is either not present or is broken.

### Note

It is recommended that out-of-tree support be fixed and used if at all possible.

It's useful to have some idea of how the tasks defined by the `autotools*` classes work and what they do behind the scenes.

- `do_configure` - Regenerates the `configure` script (using `autoreconf`) and then launches it with a standard set of arguments used during cross-compilation. You can pass additional parameters to `configure` through the `EXTRA_OECONF` or `PACKAGECONFIG_CONFGARGS` variables.
- `do_compile` - Runs `make` with arguments that specify the compiler and linker. You can pass additional arguments through the `EXTRA_OEMAKE` variable.
- `do_install` - Runs `make install` and passes in `${D}` as `DESTDIR`.

## 26.4. `base.bbclass`

The `base` class is special in that every `.bb` file implicitly inherits the class. This class contains definitions for standard basic tasks such as fetching, unpacking, configuring (empty by default), compiling (runs any `Makefile` present), installing (empty by default) and packaging (empty by default). These classes are often overridden or extended by other classes such as the `autotools` class or the `package` class.

The class also contains some commonly used functions such as `oe_runmake`, which runs `make` with the arguments specified in `EXTRA_OEMAKE` variable as well as the arguments passed directly to `oe_runmake`.

## 26.5. `bash-completion.bbclass`

Sets up packaging and dependencies appropriate for recipes that build software that includes bash-completion data.

## 26.6. `bin_package.bbclass`

The `bin_package` class is a helper class for recipes that extract the contents of a binary package (e.g. an RPM) and install those contents rather than building the binary from source. The binary package is extracted and new packages in the configured output package format are created. Extraction and installation of proprietary binaries is a good example use for this class.

### Note

For RPMs and other packages that do not contain a subdirectory, you should specify an appropriate `fetcher` parameter to point to the subdirectory. For example, if BitBake is using the `Git` fetcher (`git://`), the "subpath" parameter limits the checkout to a specific subpath of the tree. Here is an example where `${BP}` is used so that the files are extracted into the subdirectory expected by the default value of `S`:

```
SRC_URI = "git://example.com/downloads/somepackage.rpm;subpath=${BP}"
```

See the "Fetchers" section in the BitBake User Manual for more information on supported BitBake Fetchers.

## 26.7. `binconfig.bbclass`

The `binconfig` class helps to correct paths in shell scripts.

Before `pkg-config` had become widespread, libraries shipped shell scripts to give information about the libraries and include paths needed to build software (usually named `LIBNAME-config`). This class assists any recipe using such scripts.

During staging, the OpenEmbedded build system installs such scripts into the `sysroots/` directory. Inheriting this class results in all paths in these scripts being changed to point into the `sysroots/` directory so that all builds that use the script use the correct directories for the cross compiling layout. See the `BINCONFIG_GLOB` variable for more information.

## 26.8. `binconfig-disabled.bbclass`

An alternative version of the `binconfig` class, which disables binary configuration scripts by making them return an error in favor of using `pkg-config` to query the information. The scripts to be disabled should be specified using the `BINCONFIG` variable within the recipe inheriting the class.

## 26.9. `blacklist.bbclass`

The `blacklist` class prevents the OpenEmbedded build system from building specific recipes (blacklists them). To use this class, inherit the class globally and set `PNBLACKLIST` for each recipe you wish to blacklist. Specify the `PN` value as a variable flag (varflag) and provide a reason, which is reported, if the package is requested to be built as the value. For example, if you want to blacklist a recipe called "exoticware", you add the following to your `local.conf` or distribution configuration:

```
INHERIT += "blacklist"
PNBLACKLIST[exoticware] = "Not supported by our organization."
```

## 26.10. `buildhistory.bbclass`

The `buildhistory` class records a history of build output metadata, which can be used to detect possible regressions as well as used for analysis of the build output. For more information on using Build History, see the "Maintaining Build Output Quality" section in the Yocto Project Development Tasks Manual.

## 26.11. `buildstats.bbclass`

The `buildstats` class records performance statistics about each task executed during the build (e.g. elapsed time, CPU usage, and I/O usage).

When you use this class, the output goes into the `BUILDSTATS_BASE` directory, which defaults to `${TMPDIR}/buildstats/`. You can analyze the elapsed time using `scripts/pybootchartgui/pybootchartgui.py`, which produces a cascading chart of the entire build process and can be useful for highlighting bottlenecks.

Collecting build statistics is enabled by default through the `USER_CLASSES` variable from your `local.conf` file. Consequently, you do not have to do anything to enable the class. However, if you want to disable the class, simply remove "buildstats" from the `USER_CLASSES` list.

## 26.12. `buildstats-summary.bbclass`

When inherited globally, prints statistics at the end of the build on sstate re-use. In order to function, this class requires the `buildstats` class be enabled.

## 26.13. `ccache.bbclass`

The `ccache` class enables the C/C++ Compiler Cache for the build. This class is used to give a minor performance boost during the build. However, using the class can lead to unexpected side-effects. Thus, it is recommended that you do not use this class. See <http://ccache.samba.org/> for information on the C/C++ Compiler Cache.

## 26.14. `chrpath.bbclass`

The `chrpath` class is a wrapper around the "chrpath" utility, which is used during the build process for `nativesdk`, `cross`, and `cross-canadian` recipes to change `RPATH` records within binaries in order to make them relocatable.

## 26.15. `clutter.bbclass`

The `clutter` class consolidates the major and minor version naming and other common items used by Clutter and related recipes.

### Note

Unlike some other classes related to specific libraries, recipes building other software that uses Clutter do not need to inherit this class unless they use the same recipe versioning scheme that the Clutter and related recipes do.

## 26.16. `cmake.bbclass`

The `cmake` class allows for recipes that need to build software using the `CMake` build system. You can use the `EXTRA_OECMAKE` variable to specify additional configuration options to be passed using the `cmake` command line.

On the occasion that you would be installing custom CMake toolchain files supplied by the application being built, you should install them to the preferred CMake Module directory: `${D}${datadir}/cmake/Modules` during `do_install`.

### 26.17. `cm11.bbclass`

The `cm11` class provides basic support for the Linux kernel style build configuration system.

### 26.18. `compress_doc.bbclass`

Enables compression for man pages and info pages. This class is intended to be inherited globally. The default compression mechanism is `gz` (`gzip`) but you can select an alternative mechanism by setting the `DOC_COMPRESS` variable.

### 26.19. `copyleft_compliance.bbclass`

The `copyleft_compliance` class preserves source code for the purposes of license compliance. This class is an alternative to the `archiver` class and is still used by some users even though it has been deprecated in favor of the `archiver` class.

### 26.20. `copyleft_filter.bbclass`

A class used by the `archiver` and `copyleft_compliance` classes for filtering licenses. The `copyleft_filter` class is an internal class and is not intended to be used directly.

### 26.21. `core-image.bbclass`

The `core-image` class provides common definitions for the `core-image-*` image recipes, such as support for additional `IMAGE_FEATURES`.

### 26.22. `cpan*.bbclass`

The `cpan*` classes support Perl modules.

Recipes for Perl modules are simple. These recipes usually only need to point to the source's archive and then inherit the proper class file. Building is split into two methods depending on which method the module authors used.

- Modules that use old `Makefile.PL`-based build system require `cpan.bbclass` in their recipes.
- Modules that use `Build.PL`-based build system require using `cpan_build.bbclass` in their recipes.

Both build methods inherit the `cpan-base` class for basic Perl support.

### 26.23. `cross.bbclass`

The `cross` class provides support for the recipes that build the cross-compilation tools.

### 26.24. `cross-canadian.bbclass`

The `cross-canadian` class provides support for the recipes that build the Canadian Cross-compilation tools for SDKs. See the "[Cross-Development Toolchain Generation](#)" section in the Yocto Project Overview and Concepts Manual for more discussion on these cross-compilation tools.

### 26.25. `crosssdk.bbclass`

The `crosssdk` class provides support for the recipes that build the cross-compilation tools used for building SDKs. See the "[Cross-Development Toolchain Generation](#)" section in the Yocto Project Overview and Concepts Manual for more discussion on these cross-compilation tools.

### 26.26. `debian.bbclass`

The `debian` class renames output packages so that they follow the Debian naming policy (i.e. `glibc` becomes `libc6` and `glibc-devel` becomes `libc6-dev`.) Renaming includes the library name and version as part of the package name.

If a recipe creates packages for multiple libraries (shared object files of `.so` type), use the `LEAD_SONAME` variable in the recipe to specify the library on which to apply the naming scheme.

### 26.27. `deploy.bbclass`

The `deploy` class handles deploying files to the `DEPLOY_DIR_IMAGE` directory. The main function of this class is to allow the deploy step to be accelerated by shared state. Recipes that inherit this class should define their own `do_deploy` function to copy

the files to be deployed to `DEPLOYDIR`, and use `addtask` to add the task at the appropriate place, which is usually after `do_compile` or `do_install`. The class then takes care of staging the files from `DEPLOYDIR` to `DEPLOY_DIR_IMAGE`.

## 26.28. `devshell.bbclass`

The `devshell` class adds the `do_devshell` task. Distribution policy dictates whether to include this class. See the "[Using a Development Shell](#)" section in the Yocto Project Development Tasks Manual for more information about using `devshell`.

## 26.29. `devupstream.bbclass`

The `devupstream` class uses `BBCLASSEXTEND` to add a variant of the recipe that fetches from an alternative URI (e.g. Git) instead of a tarball. Following is an example:

```
BBCLASSEXTEND = "devupstream:target"
SRC_URI_class-devupstream = "git://git.example.com/example"
SRCREV_class-devupstream = "abcd1234"
```

Adding the above statements to your recipe creates a variant that has `DEFAULT_PREFERENCE` set to "-1". Consequently, you need to select the variant of the recipe to use it. Any development-specific adjustments can be done by using the `class-devupstream` override. Here is an example:

```
DEPENDS_append_class-devupstream = " gperf-native"

do_configure_prepend_class-devupstream() {
 touch ${S}/README
}
```

The class currently only supports creating a development variant of the target recipe, not `native` or `nativesdk` variants.

The `BBCLASSEXTEND` syntax (i.e. `devupstream:target`) provides support for `native` and `nativesdk` variants. Consequently, this functionality can be added in a future release.

Support for other version control systems such as Subversion is limited due to BitBake's automatic fetch dependencies (e.g. `subversion-native`).

## 26.30. `distro_features_check.bbclass`

The `distro_features_check` class allows individual recipes to check for required and conflicting `DISTRO_FEATURES`.

This class provides support for the `REQUIRED_DISTRO_FEATURES` and `CONFLICT_DISTRO_FEATURES` variables. If any conditions specified in the recipe using the above variables are not met, the recipe will be skipped.

## 26.31. `distutils*.bbclass`

The `distutils*` classes support recipes for Python version 2.x extensions, which are simple. These recipes usually only need to point to the source's archive and then inherit the proper class. Building is split into two methods depending on which method the module authors used.

- Extensions that use an Autotools-based build system require Autotools and the classes based on `distutils` in their recipes.
- Extensions that use build systems based on `distutils` require the `distutils` class in their recipes.
- Extensions that use build systems based on `setuptools` require the `setuptools` class in their recipes.

The `distutils-common-base` class is required by some of the `distutils*` classes to provide common Python2 support.

## 26.32. `distutils3*.bbclass`

The `distutils3*` classes support recipes for Python version 3.x extensions, which are simple. These recipes usually only need to point to the source's archive and then inherit the proper class. Building is split into three methods depending on which method the module authors used.

- Extensions that use an Autotools-based build system require Autotools and `distutils`-based classes in their recipes.
- Extensions that use `distutils`-based build systems require the `distutils` class in their recipes.
- Extensions that use build systems based on `setuptools3` require the `setuptools3` class in their recipes.

The `distutils3*` classes either inherit their corresponding `distutils*` class or replicate them using a Python3 version instead (e.g. `distutils3-base` inherits `distutils-common-base`, which is the same as `distutils-base` but inherits `python3native` instead of `pythonnative`).

## 26.33. `externalsrc.bbclass`

The `externalsrc` class supports building software from source code that is external to the OpenEmbedded build system. Building software from an external source tree means that the build system's normal fetch, unpack, and patch process is not used.

By default, the OpenEmbedded build system uses the `S` and `B` variables to locate unpacked recipe source code and to build it, respectively. When your recipe inherits the `externalsrc` class, you use the `EXTERNALSRC` and `EXTERNALSRC_BUILD`

variables to ultimately define `S` and `B`.

By default, this class expects the source code to support recipe builds that use the `B` variable to point to the directory in which the OpenEmbedded build system places the generated objects built from the recipes. By default, the `B` directory is set to the following, which is separate from the source directory (`S`):

```
${WORKDIR}/${BPN}/${PV}/
```

See these variables for more information: `WORKDIR`, `BPN`, and `PV`.

For more information on the `externalsrc` class, see the comments in `meta/classes/externalsrc.bbclass` in the [Source Directory](#). For information on how to use the `externalsrc` class, see the ["Building Software from an External Source"](#) section in the Yocto Project Development Tasks Manual.

## 26.34. `extrausers.bbclass`

The `extrausers` class allows additional user and group configuration to be applied at the image level. Inheriting this class either globally or from an image recipe allows additional user and group operations to be performed using the `EXTRA_USERS_PARAMS` variable.

### Note

The user and group operations added using the `extrausers` class are not tied to a specific recipe outside of the recipe for the image. Thus, the operations can be performed across the image as a whole. Use the `useradd` class to add user and group configuration to a specific recipe.

Here is an example that uses this class in an image recipe:

```
inherit extrausers
EXTRA_USERS_PARAMS = "\
 useradd -p '' tester; \
 groupadd developers; \
 userdel nobody; \
 groupdel -g video; \
 groupmod -g 1020 developers; \
 usermod -s /bin/sh tester; \
"
```

Here is an example that adds two users named "tester-jim" and "tester-sue" and assigns passwords:

```
inherit extrausers
EXTRA_USERS_PARAMS = "\
 useradd -P tester01 tester-jim; \
 useradd -P tester01 tester-sue; \
"
```

Finally, here is an example that sets the root password to "1876\*18":

```
inherit extrausers
EXTRA_USERS_PARAMS = "\
 usermod -P 1876*18 root; \
"
```

## 26.35. `fontcache.bbclass`

The `fontcache` class generates the proper post-install and post-remove (postinst and postrm) scriptlets for font packages. These scriptlets call `fc-cache` (part of `Fontconfig`) to add the fonts to the font information cache. Since the cache files are architecture-specific, `fc-cache` runs using QEMU if the postinst scriptlets need to be run on the build host during image creation.

If the fonts being installed are in packages other than the main package, set `FONT_PACKAGES` to specify the packages containing the fonts.

## 26.36. `fs-uuid.bbclass`

The `fs-uuid` class extracts UUID from `${ROOTFS}`, which must have been built by the time that this function gets called. The `fs-uuid` class only works on ext file systems and depends on `tune2fs`.

## 26.37. `gconf.bbclass`

The `gconf` class provides common functionality for recipes that need to install GConf schemas. The schemas will be put into a separate package (`${PN}-gconf`) that is created automatically when this class is inherited. This package uses the appropriate post-install and post-remove (postinst/postrm) scriptlets to register and unregister the schemas in the target image.

## 26.38. `gettext.bbclass`

The `gettext` class provides support for building software that uses the GNU `gettext` internationalization and localization system. All recipes building software that use `gettext` should inherit this class.

### 26.39. `gnomebase.bbclass`

The `gnomebase` class is the base class for recipes that build software from the GNOME stack. This class sets `SRC_URI` to download the source from the GNOME mirrors as well as extending `FILES` with the typical GNOME installation paths.

### 26.40. `gobject-introspection.bbclass`

Provides support for recipes building software that supports GObject introspection. This functionality is only enabled if the "gobject-introspection-data" feature is in `DISTRO_FEATURES` as well as "qemu-usermode" being in `MACHINE_FEATURES`.

#### Note

This functionality is backfilled by default and, if not applicable, should be disabled through `DISTRO_FEATURES_BACKFILL_CONSIDERED` or `MACHINE_FEATURES_BACKFILL_CONSIDERED`, respectively.

### 26.41. `grub-efi.bbclass`

The `grub-efi` class provides `grub-efi`-specific functions for building bootable images.

This class supports several variables:

- `INITRD`: Indicates list of filesystem images to concatenate and use as an initial RAM disk (initrd) (optional).
- `ROOTFS`: Indicates a filesystem image to include as the root filesystem (optional).
- `GRUB_GFXSERIAL`: Set this to "1" to have graphics and serial in the boot menu.
- `LABELS`: A list of targets for the automatic configuration.
- `APPEND`: An override list of append strings for each LABEL.
- `GRUB_OPTS`: Additional options to add to the configuration (optional). Options are delimited using semi-colon characters (;).
- `GRUB_TIMEOUT`: Timeout before executing the default LABEL (optional).

### 26.42. `gsettings.bbclass`

The `gsettings` class provides common functionality for recipes that need to install GSettings (glib) schemas. The schemas are assumed to be part of the main package. Appropriate post-install and post-remove (postinst/postrm) scriptlets are added to register and unregister the schemas in the target image.

### 26.43. `gtk-doc.bbclass`

The `gtk-doc` class is a helper class to pull in the appropriate `gtk-doc` dependencies and disable `gtk-doc`.

### 26.44. `gtk-icon-cache.bbclass`

The `gtk-icon-cache` class generates the proper post-install and post-remove (postinst/postrm) scriptlets for packages that use GTK+ and install icons. These scriptlets call `gtk-update-icon-cache` to add the fonts to GTK+'s icon cache. Since the cache files are architecture-specific, `gtk-update-icon-cache` is run using QEMU if the postinst scriptlets need to be run on the build host during image creation.

### 26.45. `gtk-immodules-cache.bbclass`

The `gtk-immodules-cache` class generates the proper post-install and post-remove (postinst/postrm) scriptlets for packages that install GTK+ input method modules for virtual keyboards. These scriptlets call `gtk-update-icon-cache` to add the input method modules to the cache. Since the cache files are architecture-specific, `gtk-update-icon-cache` is run using QEMU if the postinst scriptlets need to be run on the build host during image creation.

If the input method modules being installed are in packages other than the main package, set `GTKIMMODULES_PACKAGES` to specify the packages containing the modules.

### 26.46. `gzipnative.bbclass`

The `gzipnative` class enables the use of different native versions of `gzip` and `pigz` rather than the versions of these tools from the build host.

### 26.47. `icecc.bbclass`

The `icecc` class supports `Icccream`, which facilitates taking compile jobs and distributing them among remote machines.

The class stages directories with symlinks from `gcc` and `g++` to `icecc`, for both native and cross compilers. Depending on each configure or compile, the OpenEmbedded build system adds the directories at the head of the `PATH` list and then sets the

ICECC\_CXX and ICEC\_CC variables, which are the paths to the `g++` and `gcc` compilers, respectively.

For the cross compiler, the class creates a `tar.gz` file that contains the Yocto Project toolchain and sets `ICECC_VERSION`, which is the version of the cross-compiler used in the cross-development toolchain, accordingly.

The class handles all three different compile stages (i.e native ,cross-kernel and target) and creates the necessary environment `tar.gz` file to be used by the remote machines. The class also supports SDK generation.

If `ICECC_PATH` is not set in your `local.conf` file, then the class tries to locate the `icecc` binary using `which`. If `ICECC_ENV_EXEC` is set in your `local.conf` file, the variable should point to the `icecc-create-env` script provided by the user. If you do not point to a user-provided script, the build system uses the default script provided by the recipe `icecc-create-env-native.bb`.

## Note

This script is a modified version and not the one that comes with `icecc`.

If you do not want the Icecream distributed compile support to apply to specific recipes or classes, you can effectively "blacklist" them by listing the recipes and classes using the `ICECC_USER_PACKAGE_BL` and `ICECC_USER_CLASS_BL`, variables, respectively, in your `local.conf` file. Doing so causes the OpenEmbedded build system to handle these compilations locally.

Additionally, you can list recipes using the `ICECC_USER_PACKAGE_WL` variable in your `local.conf` file to force `icecc` to be enabled for recipes using an empty `PARALLEL_MAKE` variable.

Inheriting the `icecc` class changes all sstate signatures. Consequently, if a development team has a dedicated build system that populates `STATE_MIRRORS` and they want to reuse sstate from `STATE_MIRRORS`, then all developers and the build system need to either inherit the `icecc` class or nobody should.

At the distribution level, you can inherit the `icecc` class to be sure that all builders start with the same sstate signatures. After inheriting the class, you can then disable the feature by setting the `ICECC_DISABLED` variable to "1" as follows:

```
INHERIT_DISTRO_append = " icecc"
ICECC_DISABLED ??= "1"
```

This practice makes sure everyone is using the same signatures but also requires individuals that do want to use Icecream to enable the feature individually as follows in your `local.conf` file:

```
ICECC_DISABLED = ""
```

## 26.48. image.bbclass¶

The `image` class helps support creating images in different formats. First, the root filesystem is created from packages using one of the `rootfs*.bbclass` files (depending on the package format used) and then one or more image files are created.

- The `IMAGE_FSTYPES` variable controls the types of images to generate.
- The `IMAGE_INSTALL` variable controls the list of packages to install into the image.

For information on customizing images, see the "Customizing Images" section in the Yocto Project Development Tasks Manual. For information on how images are created, see the "Images" section in the Yocto Project Overview and Concepts Manual.

## 26.49. image-buildinfo.bbclass¶

The `image-buildinfo` class writes information to the target filesystem on `/etc/build`.

## 26.50. image\_types.bbclass¶

The `image_types` class defines all of the standard image output types that you can enable through the `IMAGE_FSTYPES` variable. You can use this class as a reference on how to add support for custom image output types.

By default, the `image` class automatically enables the `image_types` class. The `image` class uses the `IMGCLASSES` variable as follows:

```
IMGCLASSES = "rootfs_${IMAGE_PKGTYPE} image_types ${IMAGE_CLASSES}"
IMGCLASSES += "${@['populate_sdk_base', 'populate_sdk_ext']['linux' in d.getVar("SDK_OS")]}"
IMGCLASSES += "${@bb.utils.contains_any('IMAGE_FSTYPES', 'live iso hddimg', 'image-live', '', d)}"
IMGCLASSES += "${@bb.utils.contains('IMAGE_FSTYPES', 'container', 'image-container', '', d)}"
IMGCLASSES += "image_types_wic"
IMGCLASSES += "rootfs-postcommands"
IMGCLASSES += "image-postinst-intercepts"
inherit ${IMGCLASSES}
```

The `image_types` class also handles conversion and compression of images.

## Note

To build a VMware VMDK image, you need to add "wic.vmdk" to `IMAGE_FSTYPES`. This would also be similar for Virtual Box Virtual Disk Image ("vdi") and QEMU Copy On Write Version 2 ("qcow2") images.



## 26.51. `image-live.bbclass`

This class controls building "live" (i.e. HDDIMG and ISO) images. Live images contain syslinux for legacy booting, as well as the bootloader specified by `EFI_PROVIDER` if `MACHINE_FEATURES` contains "efi".

Normally, you do not use this class directly. Instead, you add "live" to `IMAGE_FSTYPES`.

## 26.52. `image-mklibs.bbclass`

The `image-mklibs` class enables the use of the `mklibs` utility during the `do_rootfs` task, which optimizes the size of libraries contained in the image.

By default, the class is enabled in the `local.conf.template` using the `USER_CLASSES` variable as follows:

```
USER_CLASSES += "buildstats image-mklibs image-prelink"
```

## 26.53. `image-prelink.bbclass`

The `image-prelink` class enables the use of the `prelink` utility during the `do_rootfs` task, which optimizes the dynamic linking of shared libraries to reduce executable startup time.

By default, the class is enabled in the `local.conf.template` using the `USER_CLASSES` variable as follows:

```
USER_CLASSES += "buildstats image-mklibs image-prelink"
```

## 26.54. `insane.bbclass`

The `insane` class adds a step to the package generation process so that output quality assurance checks are generated by the OpenEmbedded build system. A range of checks are performed that check the build's output for common problems that show up during runtime. Distribution policy usually dictates whether to include this class.

You can configure the sanity checks so that specific test failures either raise a warning or an error message. Typically, failures for new tests generate a warning. Subsequent failures for the same test would then generate an error message once the metadata is in a known and good condition. See the "QA Error and Warning Messages" Chapter for a list of all the warning and error messages you might encounter using a default configuration.

Use the `WARN_QA` and `ERROR_QA` variables to control the behavior of these checks at the global level (i.e. in your custom distro configuration). However, to skip one or more checks in recipes, you should use `INSANE_SKIP`. For example, to skip the check for symbolic link `.so` files in the main package of a recipe, add the following to the recipe. You need to realize that the package name override, in this example `${PN}`, must be used:

```
INSANE_SKIP_${PN} += "dev-so"
```

Please keep in mind that the QA checks exist in order to detect real or potential problems in the packaged output. So exercise caution when disabling these checks.

The following list shows the tests you can list with the `WARN_QA` and `ERROR_QA` variables:

- **`already-stripped`**: Checks that produced binaries have not already been stripped prior to the build system extracting debug symbols. It is common for upstream software projects to default to stripping debug symbols for output binaries. In order for debugging to work on the target using `-dbg` packages, this stripping must be disabled.
- **`arch`**: Checks the Executable and Linkable Format (ELF) type, bit size, and endianness of any binaries to ensure they match the target architecture. This test fails if any binaries do not match the type since there would be an incompatibility. The test could indicate that the wrong compiler or compiler options have been used. Sometimes software, like bootloaders, might need to bypass this check.
- **`buildpaths`**: Checks for paths to locations on the build host inside the output files. Currently, this test triggers too many false positives and thus is not normally enabled.
- **`build-deps`**: Determines if a build-time dependency that is specified through `DEPENDS`, explicit `RDEPENDS`, or task-level dependencies exists to match any runtime dependency. This determination is particularly useful to discover where runtime dependencies are detected and added during packaging. If no explicit dependency has been specified within the metadata, at the packaging stage it is too late to ensure that the dependency is built, and thus you can end up with an error when the package is installed into the image during the `do_rootfs` task because the auto-detected dependency was not satisfied. An example of this would be where the `update-rc.d` class automatically adds a dependency on the `initscripts-functions` package to packages that install an `initscript` that refers to `/etc/init.d/functions`. The recipe should really have an explicit `RDEPENDS` for the package in question on `initscripts-functions` so that the OpenEmbedded build system is able to ensure that the `initscripts` recipe is actually built and thus the `initscripts-functions` package is made available.
- **`compile-host-path`**: Checks the `do_compile` log for indications that paths to locations on the build host were used. Using such paths might result in host contamination of the build output.
- **`debug-deps`**: Checks that all packages except `-dbg` packages do not depend on `-dbg` packages, which would cause a packaging bug.
- **`debug-files`**: Checks for `.debug` directories in anything but the `-dbg` package. The debug files should all be in the `-dbg` package. Thus, anything packaged elsewhere is incorrect packaging.
- **`dep-cmp`**: Checks for invalid version comparison statements in runtime dependency relationships between packages (i.e. in `RDEPENDS`, `RRECOMMENDS`, `RSUGGESTS`, `RPROVIDES`, `REPLACES`, and `RCONFLICTS` variable values). Any invalid comparisons might trigger failures or undesirable behavior when passed to the package manager.



- **desktop**: Runs the `desktop-file-validate` program against any `.desktop` files to validate their contents against the specification for `.desktop` files.
- **dev-deps**: Checks that all packages except `-dev` or `-staticdev` packages do not depend on `-dev` packages, which would be a packaging bug.
- **dev-so**: Checks that the `.so` symbolic links are in the `-dev` package and not in any of the other packages. In general, these symlinks are only useful for development purposes. Thus, the `-dev` package is the correct location for them. Some very rare cases do exist for dynamically loaded modules where these symlinks are needed instead in the main package.
- **file-rdeps**: Checks that file-level dependencies identified by the OpenEmbedded build system at packaging time are satisfied. For example, a shell script might start with the line `#!/bin/bash`. This line would translate to a file dependency on `/bin/bash`. Of the three package managers that the OpenEmbedded build system supports, only RPM directly handles file-level dependencies, resolving them automatically to packages providing the files. However, the lack of that functionality in the other two package managers does not mean the dependencies do not still need resolving. This QA check attempts to ensure that explicitly declared `RDEPENDS` exist to handle any file-level dependency detected in packaged files.
- **files-invalid**: Checks for `FILES` variable values that contain `"/"`, which is invalid.
- **host-user-contaminated**: Checks that no package produced by the recipe contains any files outside of `/home` with a user or group ID that matches the user running BitBake. A match usually indicates that the files are being installed with an incorrect UID/GID, since target IDs are independent from host IDs. For additional information, see the section describing the `do_install` task.
- **incompatible-license**: Report when packages are excluded from being created due to being marked with a license that is in `INCOMPATIBLE_LICENSE`.
- **install-host-path**: Checks the `do_install` log for indications that paths to locations on the build host were used. Using such paths might result in host contamination of the build output.
- **installed-vs-shipped**: Reports when files have been installed within `do_install` but have not been included in any package by way of the `FILES` variable. Files that do not appear in any package cannot be present in an image later on in the build process. Ideally, all installed files should be packaged or not installed at all. These files can be deleted at the end of `do_install` if the files are not needed in any package.
- **invalid-chars**: Checks that the recipe metadata variables `DESCRIPTION`, `SUMMARY`, `LICENSE`, and `SECTION` do not contain non-UTF-8 characters. Some package managers do not support such characters.
- **invalid-packageconfig**: Checks that no undefined features are being added to `PACKAGECONFIG`. For example, any name "foo" for which the following form does not exist:
 

```
PACKAGECONFIG[foo] = "..."
```
- **la**: Checks `.la` files for any `TMPLDIR` paths. Any `.la` file containing these paths is incorrect since `libtool` adds the correct `sysroot` prefix when using the files automatically itself.
- **ldflags**: Ensures that the binaries were linked with the `LDFLAGS` options provided by the build system. If this test fails, check that the `LDFLAGS` variable is being passed to the linker command.
- **libdir**: Checks for libraries being installed into incorrect (possibly hardcoded) installation paths. For example, this test will catch recipes that install `/lib/bar.so` when `${base_libdir}` is "lib32". Another example is when recipes install `/usr/lib64/foo.so` when `${libdir}` is `/usr/lib`.
- **libexec**: Checks if a package contains files in `/usr/libexec`. This check is not performed if the `libexecdir` variable has been set explicitly to `/usr/libexec`.
- **packages-list**: Checks for the same package being listed multiple times through the `PACKAGES` variable value. Installing the package in this manner can cause errors during packaging.
- **perm-config**: Reports lines in `fs-perms.txt` that have an invalid format.
- **perm-line**: Reports lines in `fs-perms.txt` that have an invalid format.
- **perm-link**: Reports lines in `fs-perms.txt` that specify 'link' where the specified target already exists.
- **perms**: Currently, this check is unused but reserved.
- **pkgconfig**: Checks `.pc` files for any `TMPLDIR/WORKDIR` paths. Any `.pc` file containing these paths is incorrect since `pkg-config` itself adds the correct `sysroot` prefix when the files are accessed.
- **pkgname**: Checks that all packages in `PACKAGES` have names that do not contain invalid characters (i.e. characters other than 0-9, a-z, ., +, and -).
- **pkgv-undefined**: Checks to see if the `PKGVERSION` variable is undefined during `do_package`.
- **pkgvcheck**: Checks through the variables `RDEPENDS`, `RRECOMMENDS`, `RSUGGESTS`, `RCONFLICTS`, `RPROVIDES`, `RREPLACES`, `FILES`, `ALLOW_EMPTY`, `pkg_preinst`, `pkg_postinst`, `pkg_prerm` and `pkg_postrm`, and reports if there are variable sets that are not package-specific. Using these variables without a package suffix is bad practice, and might unnecessarily complicate dependencies of other packages within the same recipe or have other unintended consequences.
- **pn-overrides**: Checks that a recipe does not have a name (`PN`) value that appears in `OVERRIDES`. If a recipe is named such that its `PN` value matches something already in `OVERRIDES` (e.g. `PN` happens to be the same as `MACHINE` or `DISTRO`), it can have unexpected consequences. For example, assignments such as `FILES_${PN} = "xyz"` effectively turn into `FILES = "xyz"`.
- **rpaths**: Checks for `rpaths` in the binaries that contain build system paths such as `TMPLDIR`. If this test fails, bad `-rpath` options are being passed to the linker commands and your binaries have potential security issues.
- **split-strip**: Reports that splitting or stripping debug symbols from binaries has failed.
- **staticdev**: Checks for static library files (`*.a`) in non-`staticdev` packages.

- ***symlink-to-sysroot***: Checks for symlinks in packages that point into `TMPDIR` on the host. Such symlinks will work on the host, but are clearly invalid when running on the target.
- ***textrel***: Checks for ELF binaries that contain relocations in their `.text` sections, which can result in a performance impact at runtime. See the explanation for the `ELF binary` message for more information regarding runtime performance issues.
- ***useless-rpaths***: Checks for dynamic library load paths (rpaths) in the binaries that by default on a standard system are searched by the linker (e.g. `/lib` and `/usr/lib`). While these paths will not cause any breakage, they do waste space and are unnecessary.
- ***var-undefined***: Reports when variables fundamental to packaging (i.e. `WORKDIR`, `DEPLOY_DIR`, `D`, `PN`, and `PKGID`) are undefined during `do_package`.
- ***version-going-backwards***: If Build History is enabled, reports when a package being written out has a lower version than the previously written package under the same name. If you are placing output packages into a feed and upgrading packages on a target system using that feed, the version of a package going backwards can result in the target system not correctly upgrading to the "new" version of the package.

### Note

If you are not using runtime package management on your target system, then you do not need to worry about this situation.

- ***xorg-driver-abi***: Checks that all packages containing Xorg drivers have ABI dependencies. The `xserver-xorg` recipe provides driver ABI names. All drivers should depend on the ABI versions that they have been built against. Driver recipes that include `xorg-driver-input.inc` or `xorg-driver-video.inc` will automatically get these versions. Consequently, you should only need to explicitly add dependencies to binary driver recipes.

## 26.55. `insserv.bbclass`

The `insserv` class uses the `insserv` utility to update the order of symbolic links in `/etc/rc?.d/` within an image based on dependencies specified by LSB headers in the `init.d` scripts themselves.

## 26.56. `kernel.bbclass`

The `kernel` class handles building Linux kernels. The class contains code to build all kernel trees. All needed headers are staged into the `STAGING_KERNEL_DIR` directory to allow out-of-tree module builds using the `module` class.

This means that each built kernel module is packaged separately and inter-module dependencies are created by parsing the `modinfo` output. If all modules are required, then installing the `kernel-modules` package installs all packages with modules and various other kernel packages such as `kernel-vmlinux`.

The `kernel` class contains logic that allows you to embed an initial RAM filesystem (initramfs) image when you build the kernel image. For information on how to build an initramfs, see the "[Building an Initial RAM Filesystem \(initramfs\) Image](#)" section in the Yocto Project Development Tasks Manual.

Various other classes are used by the `kernel` and `module` classes internally including the `kernel-arch`, `module-base`, and `linux-kernel-base` classes.

## 26.57. `kernel-arch.bbclass`

The `kernel-arch` class sets the `ARCH` environment variable for Linux kernel compilation (including modules).

## 26.58. `kernel-devicetree.bbclass`

The `kernel-devicetree` class, which is inherited by the `kernel` class, supports device tree generation.

## 26.59. `kernel-fitimage.bbclass`

The `kernel-fitimage` class provides support to pack zImages.

## 26.60. `kernel-grub.bbclass`

The `kernel-grub` class updates the boot area and the boot menu with the kernel as the priority boot mechanism while installing a RPM to update the kernel on a deployed target.

## 26.61. `kernel-module-split.bbclass`

The `kernel-module-split` class provides common functionality for splitting Linux kernel modules into separate packages.

## 26.62. `kernel-uboot.bbclass`

The `kernel-uboot` class provides support for building from vmlinux-style kernel sources.

### 26.63. `kernel-uimage.bbclass`

The `kernel-uimage` class provides support to pack `uImage`.

### 26.64. `kernel-yocto.bbclass`

The `kernel-yocto` class provides common functionality for building from `linux-yocto` style kernel source repositories.

### 26.65. `kernelsrc.bbclass`

The `kernelsrc` class sets the Linux kernel source and version.

### 26.66. `lib_package.bbclass`

The `lib_package` class supports recipes that build libraries and produce executable binaries, where those binaries should not be installed by default along with the library. Instead, the binaries are added to a separate `${PN}-bin` package to make their installation optional.

### 26.67. `libc*.bbclass`

The `libc*` classes support recipes that build packages with `libc`:

- The `libc-common` class provides common support for building with `libc`.
- The `libc-package` class supports packaging up `glibc` and `eglibc`.

### 26.68. `license.bbclass`

The `license` class provides license manifest creation and license exclusion. This class is enabled by default using the default value for the `INHERIT_DISTRO` variable.

### 26.69. `linux-kernel-base.bbclass`

The `linux-kernel-base` class provides common functionality for recipes that build out of the Linux kernel source tree. These builds goes beyond the kernel itself. For example, the `Perf` recipe also inherits this class.

### 26.70. `linuxloader.bbclass`

Provides the function `linuxloader()`, which gives the value of the dynamic loader/linker provided on the platform. This value is used by a number of other classes.

### 26.71. `logging.bbclass`

The `logging` class provides the standard shell functions used to log messages for various BitBake severity levels (i.e. `bbplain`, `bbnote`, `bbwarn`, `bberror`, `bbfatal`, and `bbdebug`).

This class is enabled by default since it is inherited by the `base` class.

### 26.72. `meta.bbclass`

The `meta` class is inherited by recipes that do not build any output packages themselves, but act as a "meta" target for building other recipes.

### 26.73. `metadata_scm.bbclass`

The `metadata_scm` class provides functionality for querying the branch and revision of a Source Code Manager (SCM) repository.

The `base` class uses this class to print the revisions of each layer before starting every build. The `metadata_scm` class is enabled by default because it is inherited by the `base` class.

### 26.74. `migrate_localcount.bbclass`

The `migrate_localcount` class verifies a recipe's `localcount` data and increments it appropriately.

### 26.75. `mime.bbclass`

The `mime` class generates the proper post-install and post-remove (`postinst/postrm`) scriptlets for packages that install MIME type files. These scriptlets call `update-mime-database` to add the MIME types to the shared database.

## 26.76. `mirrors.bbclass`

The `mirrors` class sets up some standard `MIRRORS` entries for source code mirrors. These mirrors provide a fall-back path in case the upstream source specified in `SRC_URI` within recipes is unavailable.

This class is enabled by default since it is inherited by the `base` class.

## 26.77. `module.bbclass`

The `module` class provides support for building out-of-tree Linux kernel modules. The class inherits the `module-base` and `kernel-module-split` classes, and implements the `do_compile` and `do_install` tasks. The class provides everything needed to build and package a kernel module.

For general information on out-of-tree Linux kernel modules, see the "[Incorporating Out-of-Tree Modules](#)" section in the Yocto Project Linux Kernel Development Manual.

## 26.78. `module-base.bbclass`

The `module-base` class provides the base functionality for building Linux kernel modules. Typically, a recipe that builds software that includes one or more kernel modules and has its own means of building the module inherits this class as opposed to inheriting the `module` class.

## 26.79. `multilib*.bbclass`

The `multilib*` classes provide support for building libraries with different target optimizations or target architectures and installing them side-by-side in the same image.

For more information on using the Multilib feature, see the "[Combining Multiple Versions of Library Files into One Image](#)" section in the Yocto Project Development Tasks Manual.

## 26.80. `native.bbclass`

The `native` class provides common functionality for recipes that build tools to run on the build host (i.e. tools that use the compiler or other tools from the build host).

You can create a recipe that builds tools that run natively on the host a couple different ways:

- Create a `myrecipe-native.bb` recipe that inherits the `native` class. If you use this method, you must order the `inherit` statement in the recipe after all other `inherit` statements so that the `native` class is inherited last.

### Warning

When creating a recipe this way, the recipe name must follow this naming convention:

```
myrecipe-native.bb
```

Not using this naming convention can lead to subtle problems caused by existing code that depends on that naming convention.

- Create or modify a target recipe that contains the following:

```
BBCLASSEXTEND = "native"
```

Inside the recipe, use `_class-native` and `_class-target` overrides to specify any functionality specific to the respective native or target case.

Although applied differently, the `native` class is used with both methods. The advantage of the second method is that you do not need to have two separate recipes (assuming you need both) for native and target. All common parts of the recipe are automatically shared.

## 26.81. `nativesdk.bbclass`

The `nativesdk` class provides common functionality for recipes that wish to build tools to run as part of an SDK (i.e. tools that run on `SDKMACHINE`).

You can create a recipe that builds tools that run on the SDK machine a couple different ways:

- Create a `nativesdk-myrecipe.bb` recipe that inherits the `nativesdk` class. If you use this method, you must order the `inherit` statement in the recipe after all other `inherit` statements so that the `nativesdk` class is inherited last.
- Create a `nativesdk` variant of any recipe by adding the following:

```
BBCLASSEXTEND = "nativesdk"
```

Inside the recipe, use `_class-nativesdk` and `_class-target` overrides to specify any functionality specific to the respective SDK machine or target case.

## Warning

When creating a recipe, you must follow this naming convention:

```
nativesdk-myrecipe.bb
```

Not doing so can lead to subtle problems because code exists that depends on the naming convention.

Although applied differently, the `nativesdk` class is used with both methods. The advantage of the second method is that you do not need to have two separate recipes (assuming you need both) for the SDK machine and the target. All common parts of the recipe are automatically shared.

### 26.82. `nopackages.bbclass`

Disables packaging tasks for those recipes and classes where packaging is not needed.

### 26.83. `npm.bbclass`

Provides support for building Node.js software fetched using the [node package manager \(NPM\)](#).

## Note

Currently, recipes inheriting this class must use the `npm://` fetcher to have dependencies fetched and packaged automatically.

For information on how to create NPM packages, see the "[Creating Node Package Manager \(NPM\) Packages](#)" section in the Yocto Project Development Tasks Manual.

### 26.84. `oelint.bbclass`

The `oelint` class is an obsolete lint checking tool that exists in `meta/classes` in the [Source Directory](#).

A number of classes exist that could be generally useful in OE-Core but are never actually used within OE-Core itself. The `oelint` class is one such example. However, being aware of this class can reduce the proliferation of different versions of similar classes across multiple layers.

### 26.85. `own-mirrors.bbclass`

The `own-mirrors` class makes it easier to set up your own [PREMIRRORS](#) from which to first fetch source before attempting to fetch it from the upstream specified in [SRC\\_URI](#) within each recipe.

To use this class, inherit it globally and specify [SOURCE\\_MIRROR\\_URL](#). Here is an example:

```
INHERIT += "own-mirrors"
SOURCE_MIRROR_URL = "http://example.com/my-source-mirror"
```

You can specify only a single URL in [SOURCE\\_MIRROR\\_URL](#).

### 26.86. `package.bbclass`

The `package` class supports generating packages from a build's output. The core generic functionality is in `package.bbclass`. The code specific to particular package types resides in these package-specific classes: [package\\_deb](#), [package\\_rpm](#), [package\\_ipk](#), and [package\\_tar](#).

## Warning

The `package_tar` class is broken and not supported. It is recommended that you do not use this class.

You can control the list of resulting package formats by using the [PACKAGE\\_CLASSES](#) variable defined in your `conf/local.conf` configuration file, which is located in the [Build Directory](#). When defining the variable, you can specify one or more package types. Since images are generated from packages, a packaging class is needed to enable image generation. The first class listed in this variable is used for image generation.

If you take the optional step to set up a repository (package feed) on the development host that can be used by DNF, you can install packages from the feed while you are running the image on the target (i.e. runtime installation of packages). For more information, see the "[Using Runtime Package Management](#)" section in the Yocto Project Development Tasks Manual.

The package-specific class you choose can affect build-time performance and has space ramifications. In general, building a package with IPK takes about thirty percent less time as compared to using RPM to build the same or similar package. This comparison takes into account a complete build of the package with all dependencies previously built. The reason for this discrepancy is because the RPM package manager creates and processes more [Metadata](#) than the IPK package manager. Consequently, you might consider setting [PACKAGE\\_CLASSES](#) to "package-ipk" if you are building smaller systems.

Before making your package manager decision, however, you should consider some further things about using RPM:

- RPM starts to provide more abilities than IPK due to the fact that it processes more Metadata. For example, this information includes individual file types, file checksum generation and evaluation on install, sparse file support, conflict detection and resolution for Multilib systems, ACID style upgrade, and repackaging abilities for rollbacks.
- For smaller systems, the extra space used for the Berkeley Database and the amount of metadata when using RPM can affect your ability to perform on-device upgrades.

You can find additional information on the effects of the package class at these two Yocto Project mailing list links:

- <https://lists.yoctoproject.org/pipermail/poky/2011-May/006362.html>
- <https://lists.yoctoproject.org/pipermail/poky/2011-May/006363.html>

## 26.87. `package_deb.bbclass`

The `package_deb` class provides support for creating packages that use the Debian (i.e. `.deb`) file format. The class ensures the packages are written out in a `.deb` file format to the `${DEPLOY_DIR_DEB}` directory.

This class inherits the `package` class and is enabled through the `PACKAGE_CLASSES` variable in the `local.conf` file.

## 26.88. `package_ipk.bbclass`

The `package_ipk` class provides support for creating packages that use the IPK (i.e. `.ipk`) file format. The class ensures the packages are written out in a `.ipk` file format to the `${DEPLOY_DIR_IPK}` directory.

This class inherits the `package` class and is enabled through the `PACKAGE_CLASSES` variable in the `local.conf` file.

## 26.89. `package_rpm.bbclass`

The `package_rpm` class provides support for creating packages that use the RPM (i.e. `.rpm`) file format. The class ensures the packages are written out in a `.rpm` file format to the `${DEPLOY_DIR_RPM}` directory.

This class inherits the `package` class and is enabled through the `PACKAGE_CLASSES` variable in the `local.conf` file.

## 26.90. `package_tar.bbclass`

The `package_tar` class provides support for creating tarballs. The class ensures the packages are written out in a tarball format to the `${DEPLOY_DIR_TAR}` directory.

This class inherits the `package` class and is enabled through the `PACKAGE_CLASSES` variable in the `local.conf` file.

### Note

You cannot specify the `package_tar` class first using the `PACKAGE_CLASSES` variable. You must use `.deb`, `.ipk`, or `.rpm` file formats for your image or SDK.

## 26.91. `packagedata.bbclass`

The `packagedata` class provides common functionality for reading `pkgdata` files found in `PKGDATA_DIR`. These files contain information about each output package produced by the OpenEmbedded build system.

This class is enabled by default because it is inherited by the `package` class.

## 26.92. `packagegroup.bbclass`

The `packagegroup` class sets default values appropriate for package group recipes (e.g. `PACKAGES`, `PACKAGE_ARCH`, `ALLOW_EMPTY`, and so forth). It is highly recommended that all package group recipes inherit this class.

For information on how to use this class, see the "[Customizing Images Using Custom Package Groups](#)" section in the Yocto Project Development Tasks Manual.

Previously, this class was called the `task` class.

## 26.93. `patch.bbclass`

The `patch` class provides all functionality for applying patches during the `do_patch` task.

This class is enabled by default because it is inherited by the `base` class.

## 26.94. `perlnative.bbclass`

When inherited by a recipe, the `perlnative` class supports using the native version of Perl built by the build system rather than using the version provided by the build host.

## 26.95. `pixbufcache.bbclass`

The `pixbufcache` class generates the proper post-install and post-remove (`postinst/postrm`) scriptlets for packages that install `pixbuf` loaders, which are used with `gdk-pixbuf`. These scriptlets call `update_pixbuf_cache` to add the `pixbuf` loaders to the cache. Since the cache files are architecture-specific, `update_pixbuf_cache` is run using QEMU if the `postinst` scriptlets need to be run on the build host during image creation.

If the `pixbuf` loaders being installed are in packages other than the recipe's main package, set `PIXBUF_PACKAGES` to specify the packages containing the loaders.

## 26.96. `pkgconfig.bbclass`

The `pkgconfig` class provides a standard way to get header and library information by using `pkg-config`. This class aims to smooth integration of `pkg-config` into libraries that use it.

During staging, BitBake installs `pkg-config` data into the `sysroots/` directory. By making use of `sysroot` functionality within `pkg-config`, the `pkgconfig` class no longer has to manipulate the files.

## 26.97. `populate_sdk.bbclass`

The `populate_sdk` class provides support for SDK-only recipes. For information on advantages gained when building a cross-development toolchain using the `do_populate_sdk` task, see the "[Building an SDK Installer](#)" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

## 26.98. `populate_sdk_*.bbclass`

The `populate_sdk_*` classes support SDK creation and consist of the following classes:

- **`populate_sdk_base`**: The base class supporting SDK creation under all package managers (i.e. DEB, RPM, and opkg).
- **`populate_sdk_deb`**: Supports creation of the SDK given the Debian package manager.
- **`populate_sdk_rpm`**: Supports creation of the SDK given the RPM package manager.
- **`populate_sdk_ipk`**: Supports creation of the SDK given the opkg (IPK format) package manager.
- **`populate_sdk_ext`**: Supports extensible SDK creation under all package managers.

The `populate_sdk_base` class inherits the appropriate `populate_sdk_*` (i.e. `deb`, `rpm`, and `ipk`) based on `IMAGE_PKGTYPE`.

The base class ensures all source and destination directories are established and then populates the SDK. After populating the SDK, the `populate_sdk_base` class constructs two `sysroots`: `${SDK_ARCH}-nativesdk`, which contains the cross-compiler and associated tooling, and the target, which contains a target root filesystem that is configured for the SDK usage. These two images reside in `SDK_OUTPUT`, which consists of the following:

```
${SDK_OUTPUT}/${SDK_ARCH}-nativesdk-pkgs
${SDK_OUTPUT}/${SDKTARGETSYSROOT}/target-pkgs
```

Finally, the base `populate SDK` class creates the toolchain environment setup script, the tarball of the SDK, and the installer.

The respective `populate_sdk_deb`, `populate_sdk_rpm`, and `populate_sdk_ipk` classes each support the specific type of SDK. These classes are inherited by and used with the `populate_sdk_base` class.

For more information on the cross-development toolchain generation, see the "[Cross-Development Toolchain Generation](#)" section in the Yocto Project Overview and Concepts Manual. For information on advantages gained when building a cross-development toolchain using the `do_populate_sdk` task, see the "[Building an SDK Installer](#)" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

## 26.99. `prexport.bbclass`

The `prexport` class provides functionality for exporting `PR` values.

### Note

This class is not intended to be used directly. Rather, it is enabled when using `"bitbake-prserv-tool export"`.

## 26.100. `primport.bbclass`

The `primport` class provides functionality for importing `PR` values.

### Note

This class is not intended to be used directly. Rather, it is enabled when using `"bitbake-prserv-tool import"`.

### 26.101. `prserv.bbclass`

The `prserv` class provides functionality for using a `PR_service` in order to automatically manage the incrementing of the `PR` variable for each recipe.

This class is enabled by default because it is inherited by the `package` class. However, the OpenEmbedded build system will not enable the functionality of this class unless `PRSERV_HOST` has been set.

### 26.102. `pptest.bbclass`

The `pptest` class provides functionality for packaging and installing runtime tests for recipes that build software that provides these tests.

This class is intended to be inherited by individual recipes. However, the class' functionality is largely disabled unless "ptest" appears in `DISTRO_FEATURES`. See the "[Testing Packages With ptest](#)" section in the Yocto Project Development Tasks Manual for more information on ptest.

### 26.103. `pptest-gnome.bbclass`

Enables package tests (ptests) specifically for GNOME packages, which have tests intended to be executed with `gnome-desktop-testing`.

For information on setting up and running ptests, see the "[Testing Packages With ptest](#)" section in the Yocto Project Development Tasks Manual.

### 26.104. `python-dir.bbclass`

The `python-dir` class provides the base version, location, and site package location for Python.

### 26.105. `python3native.bbclass`

The `python3native` class supports using the native version of Python 3 built by the build system rather than support of the version provided by the build host.

### 26.106. `pythonnative.bbclass`

When inherited by a recipe, the `pythonnative` class supports using the native version of Python built by the build system rather than using the version provided by the build host.

### 26.107. `qemu.bbclass`

The `qemu` class provides functionality for recipes that either need QEMU or test for the existence of QEMU. Typically, this class is used to run programs for a target system on the build host using QEMU's application emulation mode.

### 26.108. `recipe_sanity.bbclass`

The `recipe_sanity` class checks for the presence of any host system recipe prerequisites that might affect the build (e.g. variables that are set or software that is present).

### 26.109. `relocatable.bbclass`

The `relocatable` class enables relocation of binaries when they are installed into the sysroot.

This class makes use of the `chrpath` class and is used by both the `cross` and `native` classes.

### 26.110. `remove-libtool.bbclass`

The `remove-libtool` class adds a post function to the `do_install` task to remove all `.la` files installed by `libtool`. Removing these files results in them being absent from both the sysroot and target packages.

If a recipe needs the `.la` files to be installed, then the recipe can override the removal by setting `REMOVE_LIBTOOL_LA` to "0" as follows:

```
REMOVE_LIBTOOL_LA = "0"
```

#### Note

The `remove-libtool` class is not enabled by default.



### 26.111. `report-error.bbclass`

The `report-error` class supports enabling the [error reporting tool](#), which allows you to submit build error information to a central database.

The class collects debug information for recipe, recipe version, task, machine, distro, build system, target system, host distro, branch, commit, and log. From the information, report files using a JSON format are created and stored in `${LOG_DIR}/error-report`.

### 26.112. `rm_work.bbclass`

The `rm_work` class supports deletion of temporary workspace, which can ease your hard drive demands during builds.

The OpenEmbedded build system can use a substantial amount of disk space during the build process. A portion of this space is the work files under the `${TMPDIR}/work` directory for each recipe. Once the build system generates the packages for a recipe, the work files for that recipe are no longer needed. However, by default, the build system preserves these files for inspection and possible debugging purposes. If you would rather have these files deleted to save disk space as the build progresses, you can enable `rm_work` by adding the following to your `local.conf` file, which is found in the [Build Directory](#).

```
INHERIT += "rm_work"
```

If you are modifying and building source code out of the work directory for a recipe, enabling `rm_work` will potentially result in your changes to the source being lost. To exclude some recipes from having their work directories deleted by `rm_work`, you can add the names of the recipe or recipes you are working on to the `RM_WORK_EXCLUDE` variable, which can also be set in your `local.conf` file. Here is an example:

```
RM_WORK_EXCLUDE += "busybox glibc"
```

### 26.113. `rootfs*.bbclass`

The `rootfs*` classes support creating the root filesystem for an image and consist of the following classes:

- The `rootfs-postcommands` class, which defines filesystem post-processing functions for image recipes.
- The `rootfs_deb` class, which supports creation of root filesystems for images built using `.deb` packages.
- The `rootfs_rpm` class, which supports creation of root filesystems for images built using `.rpm` packages.
- The `rootfs_ipk` class, which supports creation of root filesystems for images built using `.ipk` packages.
- The `rootfsdebugfiles` class, which installs additional files found on the build host directly into the root filesystem.

The root filesystem is created from packages using one of the `rootfs*.bbclass` files as determined by the `PACKAGE_CLASSES` variable.

For information on how root filesystem images are created, see the "[Image Generation](#)" section in the Yocto Project Overview and Concepts Manual.

### 26.114. `sanity.bbclass`

The `sanity` class checks to see if prerequisite software is present on the host system so that users can be notified of potential problems that might affect their build. The class also performs basic user configuration checks from the `local.conf` configuration file to prevent common mistakes that cause build failures. Distribution policy usually determines whether to include this class.

### 26.115. `scons.bbclass`

The `scons` class supports recipes that need to build software that uses the SCons build system. You can use the `EXTRA_OESCONS` variable to specify additional configuration options you want to pass SCons command line.

### 26.116. `SDL.bbclass`

The `SDL` class supports recipes that need to build software that uses the Simple DirectMedia Layer (SDL) library.

### 26.117. `setuptools.bbclass`

The `setuptools` class supports Python version 2.x extensions that use build systems based on `setuptools`. If your recipe uses these build systems, the recipe needs to inherit the `setuptools` class.

### 26.118. `setuptools3.bbclass`

The `setuptools3` class supports Python version 3.x extensions that use build systems based on `setuptools3`. If your recipe uses these build systems, the recipe needs to inherit the `setuptools3` class.

### 26.119. `sign_rpm.bbclass`

The `sign_rpm` class supports generating signed RPM packages.

### 26.120. `sip.bbclass`

The `sip` class supports recipes that build or package SIP-based Python bindings.

### 26.121. `siteconfig.bbclass`

The `siteconfig` class provides functionality for handling site configuration. The class is used by the `autotools` class to accelerate the `do_configure` task.

### 26.122. `siteinfo.bbclass`

The `siteinfo` class provides information about the targets that might be needed by other classes or recipes.

As an example, consider Autotools, which can require tests that must execute on the target hardware. Since this is not possible in general when cross compiling, site information is used to provide cached test results so these tests can be skipped over but still make the correct values available. The `meta/site_directory` contains test results sorted into different categories such as architecture, endianness, and the `libc` used. Site information provides a list of files containing data relevant to the current build in the `CONFIG_SITE` variable that Autotools automatically picks up.

The class also provides variables like `SITEINFO_ENDIANNESS` and `SITEINFO_BITS` that can be used elsewhere in the metadata.

### 26.123. `spdx.bbclass`

The `spdx` class integrates real-time license scanning, generation of SPDX standard output, and verification of license information during the build.

#### Note

This class is currently at the prototype stage in the 1.6 release.

### 26.124. `sstate.bbclass`

The `sstate` class provides support for Shared State (`sstate`). By default, the class is enabled through the `INHERIT_DISTRO` variable's default value.

For more information on `sstate`, see the "[Shared State Cache](#)" section in the Yocto Project Overview and Concepts Manual.

### 26.125. `staging.bbclass`

The `staging` class installs files into individual recipe work directories for sysroots. The class contains the following key tasks:

- The `do_populate_sysroot` task, which is responsible for handing the files that end up in the recipe sysroots.
- The `do_prepare_recipe_sysroot` task (a "partner" task to the `populate_sysroot` task), which installs the files into the individual recipe work directories (i.e. `WORKDIR`).

The code in the `staging` class is complex and basically works in two stages:

- **Stage One:** The first stage addresses recipes that have files they want to share with other recipes that have dependencies on the originating recipe. Normally these dependencies are installed through the `do_install` task into `{D}`. The `do_populate_sysroot` task copies a subset of these files into `{SYSROOT_DESTDIR}`. This subset of files is controlled by the `SYSROOT_DIRS`, `SYSROOT_DIRS_NATIVE`, and `SYSROOT_DIRS_BLACKLIST` variables.

#### Note

Additionally, a recipe can customize the files further by declaring a processing function in the `SYSROOT_PREPROCESS_FUNCS` variable.

A shared state (`sstate`) object is built from these files and the files are placed into a subdirectory of `tmp/sysroots-components/`. The files are scanned for hardcoded paths to the original installation location. If the location is found in text files, the hardcoded locations are replaced by tokens and a list of the files needing such replacements is created. These adjustments are referred to as "FIXMEs". The list of files that are scanned for paths is controlled by the `SSTATE_SCAN_FILES` variable.

- **Stage Two:** The second stage addresses recipes that want to use something from another recipe and declare a dependency on that recipe through the `DEPENDS` variable. The recipe will have a `do_prepare_recipe_sysroot` task and when this task executes, it creates the `recipe-sysroot` and `recipe-sysroot-native` in the recipe work directory (i.e. `WORKDIR`). The OpenEmbedded build system creates hard links to copies of the relevant files from `sysroots-components` into the recipe work directory.

**Note**

If hard links are not possible, the build system uses actual copies.

The build system then addresses any "FIXMEs" to paths as defined from the list created in the first stage.

Finally, any files in `${bindir}` within the sysroot that have the prefix "postinst-" are executed.

**Note**

Although such sysroot post installation scripts are not recommended for general use, the files do allow some issues such as user creation and module indexes to be addressed.

Because recipes can have other dependencies outside of `DEPENDS` (e.g. `do_unpack[depends] += "tar-native:do_populate_sysroot"`), the sysroot creation function `extend_recipe_sysroot` is also added as a pre-function for those tasks whose dependencies are not through `DEPENDS` but operate similarly.

When installing dependencies into the sysroot, the code traverses the dependency graph and processes dependencies in exactly the same way as the dependencies would or would not be when installed from `sstate`. This processing means, for example, a native tool would have its native dependencies added but a target library would not have its dependencies traversed or installed. The same `sstate` dependency code is used so that builds should be identical regardless of whether `sstate` was used or not. For a closer look, see the `setscene_depvalid()` function in the `sstate` class.

The build system is careful to maintain manifests of the files it installs so that any given dependency can be installed as needed. The `sstate` hash of the installed item is also stored so that if it changes, the build system can reinstall it.

**26.126. syslinux.bbclass**

The `syslinux` class provides syslinux-specific functions for building bootable images.

The class supports the following variables:

- `INITRD`: Indicates list of filesystem images to concatenate and use as an initial RAM disk (initrd). This variable is optional.
- `ROOTFS`: Indicates a filesystem image to include as the root filesystem. This variable is optional.
- `AUTO_SYSLINUXMENU`: Enables creating an automatic menu when set to "1".
- `LABELS`: Lists targets for automatic configuration.
- `APPEND`: Lists append string overrides for each label.
- `SYSINUX_OPTS`: Lists additional options to add to the syslinux file. Semicolon characters separate multiple options.
- `SYSINUX_SPLASH`: Lists a background for the VGA boot menu when you are using the boot menu.
- `SYSINUX_DEFAULT_CONSOLE`: Set to "console=ttyX" to change kernel boot default console.
- `SYSINUX_SERIAL`: Sets an alternate serial port. Or, turns off serial when the variable is set with an empty string.
- `SYSINUX_SERIAL_TTY`: Sets an alternate "console=tty..." kernel boot argument.

**26.127. systemd.bbclass**

The `systemd` class provides support for recipes that install systemd unit files.

The functionality for this class is disabled unless you have "systemd" in `DISTRO_FEATURES`.

Under this class, the recipe or Makefile (i.e. whatever the recipe is calling during the `do_install` task) installs unit files into `${D}${systemd_unitdir}/system`. If the unit files being installed go into packages other than the main package, you need to set `SYSTEMD_PACKAGES` in your recipe to identify the packages in which the files will be installed.

You should set `SYSTEMD_SERVICE` to the name of the service file. You should also use a package name override to indicate the package to which the value applies. If the value applies to the recipe's main package, use `${PN}`. Here is an example from the `connman` recipe:

```
SYSTEMD_SERVICE_${PN} = "connman.service"
```

Services are set up to start on boot automatically unless you have set `SYSTEMD_AUTO_ENABLE` to "disable".

For more information on `systemd`, see the "Selecting an Initialization Manager" section in the Yocto Project Development Tasks Manual.

**26.128. systemd-boot.bbclass**

The `systemd-boot` class provides functions specific to the systemd-boot bootloader for building bootable images. This is an internal class and is not intended to be used directly.

**Note**

The `systemd-boot` class is a result from merging the `gummiboot` class used in previous Yocto Project releases with the `systemd` project.

Set the `EFI_PROVIDER` variable to "systemd-boot" to use this class. Doing so creates a standalone EFI bootloader that is not dependent on `systemd`.

For information on more variables used and supported in this class, see the `SYSTEMD_BOOT_CFG`, `SYSTEMD_BOOT_ENTRIES`, and `SYSTEMD_BOOT_TIMEOUT` variables.

You can also see the [Systemd-boot documentation](#) for more information.

### 26.129. `terminal.bbclass`

The `terminal` class provides support for starting a terminal session. The `OE_TERMINAL` variable controls which terminal emulator is used for the session.

Other classes use the `terminal` class anywhere a separate terminal session needs to be started. For example, the `patch` class assuming `PATCHRESOLVE` is set to "user", the `cml1` class, and the `devshell` class all use the `terminal` class.

### 26.130. `testimage*.bbclass`

The `testimage*` classes support running automated tests against images using QEMU and on actual hardware. The classes handle loading the tests and starting the image. To use the classes, you need to perform steps to set up the environment.

#### Tip

Best practices include using `IMAGE_CLASSES` rather than `INHERIT` to inherit the `testimage` class for automated image testing.

The tests are commands that run on the target system over `ssh`. Each test is written in Python and makes use of the `unittest` module.

The `testimage.bbclass` runs tests on an image when called using the following:

```
$ bitbake -c testimage image
```

The `testimage-auto` class runs tests on an image after the image is constructed (i.e. `TESTIMAGE_AUTO` must be set to "1").

For information on how to enable, run, and create new tests, see the "Performing Automated Runtime Testing" section in the Yocto Project Development Tasks Manual.

### 26.131. `testsdk.bbclass`

This class supports running automated tests against software development kits (SDKs). The `testsdk` class runs tests on an SDK when called using the following:

```
$ bitbake -c testsdk image
```

#### Tip

Best practices include using `IMAGE_CLASSES` rather than `INHERIT` to inherit the `testsdk` class for automated SDK testing.

### 26.132. `texinfo.bbclass`

This class should be inherited by recipes whose upstream packages invoke the `texinfo` utilities at build-time. Native and cross recipes are made to use the dummy scripts provided by `texinfo-dummy-native`, for improved performance. Target architecture recipes use the genuine Texinfo utilities. By default, they use the Texinfo utilities on the host system.

#### Note

If you want to use the Texinfo recipe shipped with the build system, you can remove "texinfo-native" from `ASSUME_PROVIDED` and makeinfo from `SANITY_REQUIRED_UTILITIES`.

### 26.133. `tinderclient.bbclass`

The `tinderclient` class submits build results to an external Tinderbox instance.

**Note**

This class is currently unmaintained.

**26.134. toaster.bbclass¶**

The `toaster` class collects information about packages and images and sends them as events that the BitBake user interface can receive. The class is enabled when the Toaster user interface is running.

This class is not intended to be used directly.

**26.135. toolchain-scripts.bbclass¶**

The `toolchain-scripts` class provides the scripts used for setting up the environment for installed SDKs.

**26.136. typecheck.bbclass¶**

The `typecheck` class provides support for validating the values of variables set at the configuration level against their defined types. The OpenEmbedded build system allows you to define the type of a variable using the "type" varflag. Here is an example:

```
IMAGE_FEATURES[type] = "list"
```

**26.137. uboot-config.bbclass¶**

The `uboot-config` class provides support for U-Boot configuration for a machine. Specify the machine in your recipe as follows:

```
UBOOT_CONFIG ??= <default>
UBOOT_CONFIG[foo] = "config,images"
```

You can also specify the machine using this method:

```
UBOOT_MACHINE = "config"
```

See the `UBOOT_CONFIG` and `UBOOT_MACHINE` variables for additional information.

**26.138. uninative.bbclass¶**

Attempts to isolate the build system from the host distribution's C library in order to make re-use of native shared state artifacts across different host distributions practical. With this class enabled, a tarball containing a pre-built C library is downloaded at the start of the build. In the Poky reference distribution this is enabled by default through `meta/conf/distro/include/yocto-uninative.inc`. Other distributions that do not derive from poky can also "require `conf/distro/include/yocto-uninative.inc`" to use this. Alternatively if you prefer, you can build the uninative-tarball recipe yourself, publish the resulting tarball (e.g. via HTTP) and set `UNINATIVE_URL` and `UNINATIVE_CHECKSUM` appropriately. For an example, see the `meta/conf/distro/include/yocto-uninative.inc`.

The `uninative` class is also used unconditionally by the extensible SDK. When building the extensible SDK, `uninative-tarball` is built and the resulting tarball is included within the SDK.

**26.139. update-alternatives.bbclass¶**

The `update-alternatives` class helps the alternatives system when multiple sources provide the same command. This situation occurs when several programs that have the same or similar function are installed with the same name. For example, the `ar` command is available from the `busybox`, `binutils` and `elfutils` packages. The `update-alternatives` class handles renaming the binaries so that multiple packages can be installed without conflicts. The `ar` command still works regardless of which packages are installed or subsequently removed. The class renames the conflicting binary in each package and symlinks the highest priority binary during installation or removal of packages.

To use this class, you need to define a number of variables:

- `ALTERNATIVE`
- `ALTERNATIVE_LINK_NAME`
- `ALTERNATIVE_TARGET`
- `ALTERNATIVE_PRIORITY`

These variables list alternative commands needed by a package, provide pathnames for links, default links for targets, and so forth. For details on how to use this class, see the comments in the `update-alternatives.bbclass` file.

**Note**

You can use the `update-alternatives` command directly in your recipes. However, this class simplifies things in most cases.

## 26.140. `update-rc.d.bbclass`

The `update-rc.d` class uses `update-rc.d` to safely install an initialization script on behalf of the package. The OpenEmbedded build system takes care of details such as making sure the script is stopped before a package is removed and started when the package is installed.

Three variables control this class: `INITSCRIPT_PACKAGES`, `INITSCRIPT_NAME` and `INITSCRIPT_PARAMS`. See the variable links for details.

## 26.141. `useradd*.bbclass`

The `useradd*` classes support the addition of users or groups for usage by the package on the target. For example, if you have packages that contain system services that should be run under their own user or group, you can use these classes to enable creation of the user or group. The `meta-skeleton/recipes-skeleton/useradd/useradd-example.bb` recipe in the [Source Directory](#) provides a simple example that shows how to add three users and groups to two packages. See the `useradd-example.bb` recipe for more information on how to use these classes.

The `useradd_base` class provides basic functionality for user or groups settings.

The `useradd*` classes support the `USERADD_PACKAGES`, `USERADD_PARAM`, `GROUPADD_PARAM`, and `GROUPMEMS_PARAM` variables.

The `useradd-staticids` class supports the addition of users or groups that have static user identification (`uid`) and group identification (`gid`) values.

The default behavior of the OpenEmbedded build system for assigning `uid` and `gid` values when packages add users and groups during package install time is to add them dynamically. This works fine for programs that do not care what the values of the resulting users and groups become. In these cases, the order of the installation determines the final `uid` and `gid` values. However, if non-deterministic `uid` and `gid` values are a problem, you can override the default, dynamic application of these values by setting static values. When you set static values, the OpenEmbedded build system looks in `BBPATH` for `files/passwd` and `files/group` files for the values.

To use static `uid` and `gid` values, you need to set some variables. See the `USERADDEXTENSION`, `USERADD_UID_TABLES`, `USERADD_GID_TABLES`, and `USERADD_ERROR_DYNAMIC` variables. You can also see the `useradd` class for additional information.

### Notes

You do not use the `useradd-staticids` class directly. You either enable or disable the class by setting the `USERADDEXTENSION` variable. If you enable or disable the class in a configured system, `TMPDIR` might contain incorrect `uid` and `gid` values. Deleting the `TMPDIR` directory will correct this condition.

## 26.142. `utility-tasks.bbclass`

The `utility-tasks` class provides support for various "utility" type tasks that are applicable to all recipes, such as `do_clean` and `do_listtasks`.

This class is enabled by default because it is inherited by the `base` class.

## 26.143. `utils.bbclass`

The `utils` class provides some useful Python functions that are typically used in inline Python expressions (e.g. `_${@...}`). One example use is for `bb.utils.contains()`.

This class is enabled by default because it is inherited by the `base` class.

## 26.144. `vala.bbclass`

The `vala` class supports recipes that need to build software written using the Vala programming language.

## 26.145. `waf.bbclass`

The `waf` class supports recipes that need to build software that uses the Waf build system. You can use the `EXTRA_OECONF` or `PACKAGECONFIG_CONFARGS` variables to specify additional configuration options to be passed on the Waf command line.

## Chapter 27. Tasks

### 27.1. Normal Recipe Build Tasks

#### 27.1.1. `do_build`

#### 27.1.2. `do_compile`

#### 27.1.3. `do_compile_ptest_base`

#### 27.1.4. `do_configure`

#### 27.1.5. `do_configure_ptest_base`

- [27.1.6.do\\_deploy](#)
- [27.1.7.do\\_fetch](#)
- [27.1.8.do\\_image](#)
- [27.1.9.do\\_image\\_complete](#)
- [27.1.10.do\\_install](#)
- [27.1.11.do\\_install\\_ptest\\_base](#)
- [27.1.12.do\\_package](#)
- [27.1.13.do\\_package\\_qa](#)
- [27.1.14.do\\_package\\_write\\_deb](#)
- [27.1.15.do\\_package\\_write\\_ipk](#)
- [27.1.16.do\\_package\\_write\\_rpm](#)
- [27.1.17.do\\_package\\_write\\_tar](#)
- [27.1.18.do\\_packagedata](#)
- [27.1.19.do\\_patch](#)
- [27.1.20.do\\_populate\\_lic](#)
- [27.1.21.do\\_populate\\_sdk](#)
- [27.1.22.do\\_populate\\_sysroot](#)
- [27.1.23.do\\_prepare\\_recipe\\_sysroot](#)
- [27.1.24.do\\_rm\\_work](#)
- [27.1.25.do\\_unpack](#)
- 27.2. Manually Called Tasks**
  - [27.2.1.do\\_checkpkg](#)
  - [27.2.2.do\\_checkuri](#)
  - [27.2.3.do\\_clean](#)
  - [27.2.4.do\\_cleanall](#)
  - [27.2.5.do\\_cleansstate](#)
  - [27.2.6.do\\_devpyshell](#)
  - [27.2.7.do\\_devshell](#)
  - [27.2.8.do\\_listtasks](#)
  - [27.2.9.do\\_package\\_index](#)
- 27.3. Image-Related Tasks**
  - [27.3.1.do\\_bootimg](#)
  - [27.3.2.do\\_bundle\\_initramfs](#)
  - [27.3.3.do\\_rootfs](#)
  - [27.3.4.do\\_testimage](#)
  - [27.3.5.do\\_testimage\\_auto](#)
- 27.4. Kernel-Related Tasks**
  - [27.4.1.do\\_compile\\_kernelmodules](#)
  - [27.4.2.do\\_diffconfig](#)
  - [27.4.3.do\\_kernel\\_checkout](#)
  - [27.4.4.do\\_kernel\\_configcheck](#)
  - [27.4.5.do\\_kernel\\_configme](#)
  - [27.4.6.do\\_kernel\\_menuconfig](#)
  - [27.4.7.do\\_kernel\\_metadata](#)
  - [27.4.8.do\\_menuconfig](#)
  - [27.4.9.do\\_savedefconfig](#)
  - [27.4.10.do\\_shared\\_workdir](#)
  - [27.4.11.do\\_sizecheck](#)
  - [27.4.12.do\\_strip](#)
  - [27.4.13.do\\_validate\\_branches](#)
- 27.5. Miscellaneous Tasks**
  - [27.5.1.do\\_spdx](#)

Tasks are units of execution for BitBake. Recipes ( `.bb` files) use tasks to complete configuring, compiling, and packaging software. This chapter provides a reference of the tasks defined in the OpenEmbedded build system.

## 27.1. Normal Recipe Build Tasks¶

The following sections describe normal tasks associated with building a recipe. For more information on tasks and dependencies, see the "[Tasks](#)" and "[Dependencies](#)" sections in the BitBake User Manual.

### 27.1.1. do\_build¶

The default task for all recipes. This task depends on all other normal tasks required to build a recipe.

### 27.1.2. do\_compile¶

Compiles the source code. This task runs with the current working directory set to `${B}`.

The default behavior of this task is to run the `oe_runmake` function if a makefile (Makefile, makefile, or GNUmakefile) is found. If no such file is found, the `do_compile` task does nothing.

### 27.1.3. do\_compile\_ptest\_base¶

Compiles the runtime test suite included in the software being built.

#### 27.1.4. `do_configure`

Configures the source by enabling and disabling any build-time and configuration options for the software being built. The task runs with the current working directory set to `${B}`.

The default behavior of this task is to run `oe_runmake clean` if a makefile (Makefile, makefile, or GNUmakefile) is found and `CLEANBROKEN` is not set to "1". If no such file is found or the `CLEANBROKEN` variable is set to "1", the `do_configure` task does nothing.

#### 27.1.5. `do_configure_ptest_base`

Configures the runtime test suite included in the software being built.

#### 27.1.6. `do_deploy`

Writes output files that are to be deployed to `${DEPLOY_DIR_IMAGE}`. The task runs with the current working directory set to `${B}`.

Recipes implementing this task should inherit the `deploy` class and should write the output to `${DEPLOYDIR}`, which is not to be confused with `${DEPLOY_DIR}`. The `deploy` class sets up `do_deploy` as a shared state (sstate) task that can be accelerated through sstate use. The sstate mechanism takes care of copying the output from `${DEPLOYDIR}` to `${DEPLOY_DIR_IMAGE}`.

### Caution

Do not write the output directly to `${DEPLOY_DIR_IMAGE}`, as this causes the sstate mechanism to malfunction.

The `do_deploy` task is not added as a task by default and consequently needs to be added manually. If you want the task to run after `do_compile`, you can add it by doing the following:

```
addtask deploy after do_compile
```

Adding `do_deploy` after other tasks works the same way.

### Note

You do not need to add `before do_build` to the `addtask` command (though it is harmless), because the `base` class contains the following:

```
do_build[recrdeptask] += "do_deploy"
```

See the "[Dependencies](#)" section in the BitBake User Manual for more information.

If the `do_deploy` task re-executes, any previous output is removed (i.e. "cleaned").

#### 27.1.7. `do_fetch`

Fetches the source code. This task uses the `SRC_URI` variable and the argument's prefix to determine the correct `fetcher` module.

#### 27.1.8. `do_image`

Starts the image generation process. The `do_image` task runs after the OpenEmbedded build system has run the `do_rootfs` task during which packages are identified for installation into the image and the root filesystem is created, complete with post-processing.

The `do_image` task performs pre-processing on the image through the `IMAGE_PREPROCESS_COMMAND` and dynamically generates supporting `do_image_*` tasks as needed.

For more information on image creation, see the "[Image Generation](#)" section in the Yocto Project Overview and Concepts Manual.

#### 27.1.9. `do_image_complete`

Completes the image generation process. The `do_image_complete` task runs after the OpenEmbedded build system has run the `do_image` task during which image pre-processing occurs and through dynamically generated `do_image_*` tasks the image is constructed.

The `do_image_complete` task performs post-processing on the image through the `IMAGE_POSTPROCESS_COMMAND`.

For more information on image creation, see the "[Image Generation](#)" section in the Yocto Project Overview and Concepts Manual.

#### 27.1.10. `do_install`



Copies files that are to be packaged into the holding area `${D}`. This task runs with the current working directory set to `${B}`, which is the compilation directory. The `do_install` task, as well as other tasks that either directly or indirectly depend on the installed files (e.g. `do_package`, `do_package_write_*`, and `do_rootfs`), run under `fakeroot`.

## Caution

When installing files, be careful not to set the owner and group IDs of the installed files to unintended values. Some methods of copying files, notably when using the recursive `cp` command, can preserve the UID and/or GID of the original file, which is usually not what you want. The `host-user-contaminated` QA check checks for files that probably have the wrong ownership.

Safe methods for installing files include the following:

- The `install` utility. This utility is the preferred method.
- The `cp` command with the `"--no-preserve=ownership"` option.
- The `tar` command with the `"--no-same-owner"` option. See the `bin_package.bbclass` file in the `meta/classes` directory of the [Source Directory](#) for an example.

### 27.1.11. do\_install\_ptest\_base

Copies the runtime test suite files from the compilation directory to a holding area.

### 27.1.12. do\_package

Analyzes the content of the holding area `${D}` and splits the content into subsets based on available packages and files. This task makes use of the `PACKAGES` and `FILES` variables.

The `do_package` task, in conjunction with the `do_packagedata` task, also saves some important package metadata. For additional information, see the `PKGDESTWORK` variable and the ["Automatically Added Runtime Dependencies"](#) section in the Yocto Project Overview and Concepts Manual.

### 27.1.13. do\_package\_qa

Runs QA checks on packaged files. For more information on these checks, see the [insane](#) class.

### 27.1.14. do\_package\_write\_deb

Creates Debian packages (i.e. `*.deb` files) and places them in the `${DEPLOY_DIR_DEB}` directory in the package feeds area. For more information, see the ["Package Feeds"](#) section in the Yocto Project Overview and Concepts Manual.

### 27.1.15. do\_package\_write\_ipk

Creates IPK packages (i.e. `*.ipk` files) and places them in the `${DEPLOY_DIR_IPK}` directory in the package feeds area. For more information, see the ["Package Feeds"](#) section in the Yocto Project Overview and Concepts Manual.

### 27.1.16. do\_package\_write\_rpm

Creates RPM packages (i.e. `*.rpm` files) and places them in the `${DEPLOY_DIR_RPM}` directory in the package feeds area. For more information, see the ["Package Feeds"](#) section in the Yocto Project Overview and Concepts Manual.

### 27.1.17. do\_package\_write\_tar

Creates tarballs and places them in the `${DEPLOY_DIR_TAR}` directory in the package feeds area. For more information, see the ["Package Feeds"](#) section in the Yocto Project Overview and Concepts Manual.

### 27.1.18. do\_packagedata

Saves package metadata generated by the `do_package` task in `PKGDATA_DIR` to make it available globally.

### 27.1.19. do\_patch

Locates patch files and applies them to the source code.

After fetching and unpacking source files, the build system uses the recipe's `SRC_URI` statements to locate and apply patch files to the source code.

## Note

The build system uses the `FILESPATH` variable to determine the default set of directories when searching for patches.

Patch files, by default, are `*.patch` and `*.diff` files created and kept in a subdirectory of the directory holding the recipe file. For example, consider the [bluez5](#) recipe from the OE-Core layer (i.e. `poky/meta`):

poky/meta/recipes-connectivity/bluez5

This recipe has two patch files located here:

poky/meta/recipes-connectivity/bluez5/bluez5

In the `bluez5` recipe, the `SRC_URI` statements point to the source and patch files needed to build the package.

### Note

In the case for the `bluez5_5.48.bb` recipe, the `SRC_URI` statements are from an include file `bluez5.inc`.

As mentioned earlier, the build system treats files whose file types are `.patch` and `.diff` as patch files. However, you can use the `"apply=yes"` parameter with the `SRC_URI` statement to indicate any file as a patch file:

```
SRC_URI = " \
 git://path_to_repo/some_package \
 file://file;apply=yes \
"
```

Conversely, if you have a directory full of patch files and you want to exclude some so that the `do_patch` task does not apply them during the patch phase, you can use the `"apply=no"` parameter with the `SRC_URI` statement:

```
SRC_URI = " \
 git://path_to_repo/some_package \
 file://path_to_Lots_of_patch_files \
 file://path_to_Lots_of_patch_files/patch_file5;apply=no \
"
```

In the previous example, assuming all the files in the directory holding the patch files end with either `.patch` or `.diff`, every file would be applied as a patch by default except for the `patch_file5` patch.

You can find out more about the patching process in the "[Patching](#)" section in the Yocto Project Overview and Concepts Manual and the "[Patching Code](#)" section in the Yocto Project Development Tasks Manual.

#### 27.1.20. `do_populate_lic`

Writes license information for the recipe that is collected later when the image is constructed.

#### 27.1.21. `do_populate_sdk`

Creates the file and directory structure for an installable SDK. See the "[SDK Generation](#)" section in the Yocto Project Overview and Concepts Manual for more information.

#### 27.1.22. `do_populate_sysroot`

Stages (copies) a subset of the files installed by the `do_install` task into the appropriate sysroot. For information on how to access these files from other recipes, see the `STAGING_DIR*` variables. Directories that would typically not be needed by other recipes at build time (e.g. `/etc`) are not copied by default.

For information on what directories are copied by default, see the `SYSROOT_DIRS*` variables. You can change these variables inside your recipe if you need to make additional (or fewer) directories available to other recipes at build time.

The `do_populate_sysroot` task is a shared state (sstate) task, which means that the task can be accelerated through sstate use. Realize also that if the task is re-executed, any previous output is removed (i.e. "cleaned").

#### 27.1.23. `do_prepare_recipe_sysroot`

Installs the files into the individual recipe specific sysroots (i.e. `recipe-sysroot` and `recipe-sysroot-native` under `${WORKDIR}`) based upon the dependencies specified by `DEPENDS`. See the "[staging](#)" class for more information.

#### 27.1.24. `do_rm_work`

Removes work files after the OpenEmbedded build system has finished with them. You can learn more by looking at the "[rm\\_work.bbclass](#)" section.

#### 27.1.25. `do_unpack`

Unpacks the source code into a working directory pointed to by `${WORKDIR}`. The `S` variable also plays a role in where unpacked source files ultimately reside. For more information on how source files are unpacked, see the "[Source Fetching](#)" section in the Yocto Project Overview and Concepts Manual and also see the `WORKDIR` and `S` variable descriptions.

### 27.2. Manually Called Tasks

These tasks are typically manually triggered (e.g. by using the `bitbake -c` command-line option):

#### 27.2.1. `do_checkpkg`

Provides information about the recipe including its upstream version and status. The upstream version and status reveals whether or not a version of the recipe exists upstream and a status of not updated, updated, or unknown.

To check the upstream version and status of a recipe, use the following devtool commands:

```
$ devtool latest-version
$ devtool check-upgrade-status
```

See the "[devtool Quick Reference](#)" chapter for more information on devtool. See the "[Checking on the Upgrade Status of a Recipe](#)" section for information on checking the upgrade status of a recipe.

To build the `checkpkg` task, use the `bitbake` command with the `-c` option and task name:

```
$ bitbake core-image-minimal -c checkpkg
```

By default, the results are stored in `$LOG_DIR` (e.g. `$BUILD_DIR/tmp/log`).

### 27.2.2. `do_checkuri`

Validates the `SRC_URI` value.

### 27.2.3. `do_clean`

Removes all output files for a target from the `do_unpack` task forward (i.e. `do_unpack`, `do_configure`, `do_compile`, `do_install`, and `do_package`).

You can run this task using BitBake as follows:

```
$ bitbake -c clean recipe
```

Running this task does not remove the `sstate` cache files. Consequently, if no changes have been made and the recipe is rebuilt after cleaning, output files are simply restored from the `sstate` cache. If you want to remove the `sstate` cache files for the recipe, you need to use the `do_cleansstate` task instead (i.e. `bitbake -c cleansstate recipe`).

### 27.2.4. `do_cleanall`

Removes all output files, shared state (`sstate`) cache, and downloaded source files for a target (i.e. the contents of `DL_DIR`). Essentially, the `do_cleanall` task is identical to the `do_cleansstate` task with the added removal of downloaded source files.

You can run this task using BitBake as follows:

```
$ bitbake -c cleanall recipe
```

Typically, you would not normally use the `cleanall` task. Do so only if you want to start fresh with the `do_fetch` task.

### 27.2.5. `do_cleansstate`

Removes all output files and shared state (`sstate`) cache for a target. Essentially, the `do_cleansstate` task is identical to the `do_clean` task with the added removal of shared state (`sstate`) cache.

You can run this task using BitBake as follows:

```
$ bitbake -c cleansstate recipe
```

When you run the `do_cleansstate` task, the OpenEmbedded build system no longer uses any `sstate`. Consequently, building the recipe from scratch is guaranteed.

## Note

The `do_cleansstate` task cannot remove `sstate` from a remote `sstate` mirror. If you need to build a target from scratch using remote mirrors, use the `-f` option as follows:

```
$ bitbake -f -c do_cleansstate target
```

### 27.2.6. `do_devpyshell`

Starts a shell in which an interactive Python interpreter allows you to interact with the BitBake build environment. From within this shell, you can directly examine and set bits from the data store and execute functions as if within the BitBake environment. See the "[Using a Development Python Shell](#)" section in the Yocto Project Development Tasks Manual for more information about using `devpyshell`.

### 27.2.7. `do_devshell`

Starts a shell whose environment is set up for development, debugging, or both. See the "[Using a Development Shell](#)" section in the Yocto Project Development Tasks Manual for more information about using `devshell`.

### 27.2.8. `do_listtasks`

Lists all defined tasks for a target.

### 27.2.9. `do_package_index`

Creates or updates the index in the [Package Feeds](#) area.

#### Note

This task is not triggered with the `bitbake -c` command-line option as are the other tasks in this section. Because this task is specifically for the `package-index` recipe, you run it using `bitbake package-index`.

## 27.3. Image-Related Tasks

The following tasks are applicable to image recipes.

### 27.3.1. `do_bootimg`

Creates a bootable live image. See the [IMAGE\\_FSTYPES](#) variable for additional information on live image types.

### 27.3.2. `do_bundle_initramfs`

Combines an initial RAM disk (initramfs) image and kernel together to form a single image. The [CONFIG\\_INITRAMFS\\_SOURCE](#) variable has some more information about these types of images.

### 27.3.3. `do_rootfs`

Creates the root filesystem (file and directory structure) for an image. See the "[Image Generation](#)" section in the Yocto Project Overview and Concepts Manual for more information on how the root filesystem is created.

### 27.3.4. `do_testimage`

Boots an image and performs runtime tests within the image. For information on automatically testing images, see the "[Performing Automated Runtime Testing](#)" section in the Yocto Project Development Tasks Manual.

### 27.3.5. `do_testimage_auto`

Boots an image and performs runtime tests within the image immediately after it has been built. This task is enabled when you set [TESTIMAGE\\_AUTO](#) equal to "1".

For information on automatically testing images, see the "[Performing Automated Runtime Testing](#)" section in the Yocto Project Development Tasks Manual.

## 27.4. Kernel-Related Tasks

The following tasks are applicable to kernel recipes. Some of these tasks (e.g. the `do_menuconfig` task) are also applicable to recipes that use Linux kernel style configuration such as the BusyBox recipe.

### 27.4.1. `do_compile_kernelmodules`

Runs the step that builds the kernel modules (if needed). Building a kernel consists of two steps: 1) the kernel (`vmlinux`) is built, and 2) the modules are built (i.e. `make modules`).

### 27.4.2. `do_diffconfig`

When invoked by the user, this task creates a file containing the differences between the original config as produced by `do_kernel_configme` task and the changes made by the user with other methods (i.e. using `do_kernel_menuconfig`). Once the file of differences is created, it can be used to create a config fragment that only contains the differences. You can invoke this task from the command line as follows:

```
$ bitbake linux-yocto -c diffconfig
```

For more information, see the "[Creating Configuration Fragments](#)" section in the Yocto Project Linux Kernel Development Manual.

### 27.4.3. `do_kernel_checkout`

Converts the newly unpacked kernel source into a form with which the OpenEmbedded build system can work. Because the kernel source can be fetched in several different ways, the `do_kernel_checkout` task makes sure that subsequent tasks are given a clean working tree copy of the kernel with the correct branches checked out.

### 27.4.4. `do_kernel_configcheck`

Validates the configuration produced by the `do_kernel_menuconfig` task. The `do_kernel_configcheck` task produces warnings when a requested configuration does not appear in the final `.config` file or when you override a policy configuration in a hardware configuration fragment. You can run this task explicitly and view the output by using the following command:

```
$ bitbake linux-yocto -c kernel_configcheck -f
```

For more information, see the "[Validating Configuration](#)" section in the Yocto Project Linux Kernel Development Manual.

#### 27.4.5. `do_kernel_configme`

After the kernel is patched by the `do_patch` task, the `do_kernel_configme` task assembles and merges all the kernel config fragments into a merged configuration that can then be passed to the kernel configuration phase proper. This is also the time during which user-specified defconfigs are applied if present, and where configuration modes such as `--allnoconfig` are applied.

#### 27.4.6. `do_kernel_menuconfig`

Invoked by the user to manipulate the `.config` file used to build a linux-yocto recipe. This task starts the Linux kernel configuration tool, which you then use to modify the kernel configuration.

#### Note

You can also invoke this tool from the command line as follows:

```
$ bitbake linux-yocto -c menuconfig
```

See the "[Using menuconfig](#)" section in the Yocto Project Linux Kernel Development Manual for more information on this configuration tool.

#### 27.4.7. `do_kernel_metadata`

Collects all the features required for a given kernel build, whether the features come from `SRC_URI` or from Git repositories. After collection, the `do_kernel_metadata` task processes the features into a series of config fragments and patches, which can then be applied by subsequent tasks such as `do_patch` and `do_kernel_configme`.

#### 27.4.8. `do_menuconfig`

Runs `make menuconfig` for the kernel. For information on `menuconfig`, see the "[Using menuconfig](#)" section in the Yocto Project Linux Kernel Development Manual.

#### 27.4.9. `do_savedefconfig`

When invoked by the user, creates a defconfig file that can be used instead of the default defconfig. The saved defconfig contains the differences between the default defconfig and the changes made by the user using other methods (i.e. the `do_kernel_menuconfig` task). You can invoke the task using the following command:

```
$ bitbake linux-yocto -c savedefconfig
```

#### 27.4.10. `do_shared_workdir`

After the kernel has been compiled but before the kernel modules have been compiled, this task copies files required for module builds and which are generated from the kernel build into the shared work directory. With these copies successfully copied, the `do_compile_kernelmodules` task can successfully build the kernel modules in the next step of the build.

#### 27.4.11. `do_sizecheck`

After the kernel has been built, this task checks the size of the stripped kernel image against `KERNEL_IMAGE_MAXSIZE`. If that variable was set and the size of the stripped kernel exceeds that size, the kernel build produces a warning to that effect.

#### 27.4.12. `do_strip`

If `KERNEL_IMAGE_STRIP_EXTRA_SECTIONS` is defined, this task strips the sections named in that variable from `vmlinux`. This stripping is typically used to remove nonessential sections such as `.comment` sections from a size-sensitive configuration.

#### 27.4.13. `do_validate_branches`

After the kernel is unpacked but before it is patched, this task makes sure that the machine and metadata branches as specified by the `SRCREV` variables actually exist on the specified branches. If these branches do not exist and `AUTOREV` is not being used, the `do_validate_branches` task fails during the build.

### 27.5. Miscellaneous Tasks

The following sections describe miscellaneous tasks.

#### 27.5.1. `do_spdx`

A build stage that takes the source code and scans it on a remote FOSSOLOGY server in order to produce an SPDX document. This task applies only to the `spdx` class.

## Chapter 28. devtool Quick Reference¶

[28.1. Getting Help](#)  
[28.2. The Workspace Layer Structure](#)  
[28.3. Adding a New Recipe to the Workspace Layer](#)  
[28.4. Extracting the Source for an Existing Recipe](#)  
[28.5. Synchronizing a Recipe's Extracted Source Tree](#)  
[28.6. Modifying an Existing Recipe](#)  
[28.7. Edit an Existing Recipe](#)  
[28.8. Updating a Recipe](#)  
[28.9. Checking on the Upgrade Status of a Recipe](#)  
[28.10. Upgrading a Recipe](#)  
[28.11. Resetting a Recipe](#)  
[28.12. Building Your Recipe](#)  
[28.13. Building Your Image](#)  
[28.14. Deploying Your Software on the Target Machine](#)  
[28.15. Removing Your Software from the Target Machine](#)  
[28.16. Creating the Workspace Layer in an Alternative Location](#)  
[28.17. Get the Status of the Recipes in Your Workspace](#)  
[28.18. Search for Available Target Recipes](#)

The `devtool` command-line tool provides a number of features that help you build, test, and package software. This command is available alongside the `bitbake` command. Additionally, the `devtool` command is a key part of the extensible SDK.

This chapter provides a Quick Reference for the `devtool` command. For more information on how to apply the command when using the extensible SDK, see the "[Using the Extensible SDK](#)" chapter in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

### 28.1. Getting Help¶

The `devtool` command line is organized similarly to Git in that it has a number of sub-commands for each function. You can run `devtool --help` to see all the commands:

```
$ devtool -h
NOTE: Starting bitbake server...
usage: devtool [--basepath BASEPATH] [--bbpath BBPATH] [-d] [-q]
 [--color COLOR] [-h]
 <subcommand> ...

OpenEmbedded development tool

options:
 --basepath BASEPATH Base directory of SDK / build directory
 --bbpath BBPATH Explicitly specify the BBPATH, rather than getting it
 from the metadata
 -d, --debug Enable debug output
 -q, --quiet Print only errors
 --color COLOR Colorize output (where COLOR is auto, always, never)
 -h, --help show this help message and exit

subcommands:
 Beginning work on a recipe:
 add Add a new recipe
 modify Modify the source for an existing recipe
 upgrade Upgrade an existing recipe
 Getting information:
 status Show workspace status
 search Search available recipes
 latest-version Report the latest version of an existing recipe
 check-upgrade-status Report upgradability for multiple (or all) recipes
 Working on a recipe in the workspace:
 build Build a recipe
 rename Rename a recipe file in the workspace
 edit-recipe Edit a recipe file
 find-recipe Find a recipe file
 configure-help Get help on configure script options
 update-recipe Apply changes from external source tree to recipe
 reset Remove a recipe from your workspace
 finish Finish working on a recipe in your workspace
 Testing changes on target:
 deploy-target Deploy recipe output files to live target machine
 undeploy-target Undeploy recipe output files in live target machine
 build-image Build image including workspace recipe packages
 Advanced:
 create-workspace Set up workspace in an alternative location
 export Export workspace into a tar archive
 import Import exported tar archive into workspace
 extract Extract the source for an existing recipe
 sync Synchronize the source tree for an existing recipe
 Use devtool <subcommand> --help to get help on a specific command
```

As directed in the general help output, you can get more syntax on a specific command by providing the command name and using "`--help`":

```
$ devtool add --help
NOTE: Starting bitbake server...
usage: devtool add [-h] [--same-dir | --no-same-dir] [--fetch URI]
 [--fetch-dev] [--version VERSION] [--no-git]
 [--srcrev SRCREV | --autorev] [--srcbranch SRCBRANCH]
 [--binary] [--also-native] [--src-subdir SUBDIR]
```

```
[--mirrors] [--provides PROVIDES]
[recipeName] [srcTree] [fetchUri]
```

Adds a new recipe to the workspace to build a specified source tree. Can optionally fetch a remote URI and unpack it to create the source tree.

arguments:

|            |                                                                                                                             |
|------------|-----------------------------------------------------------------------------------------------------------------------------|
| recipeName | Name for new recipe to add (just name - no version, path or extension). If not specified, will attempt to auto-detect it.   |
| srcTree    | Path to external source tree. If not specified, a subdirectory of /home/scottrif/poky/build/workspace/sources will be used. |
| fetchUri   | Fetch the specified URI and extract it to create the source tree                                                            |

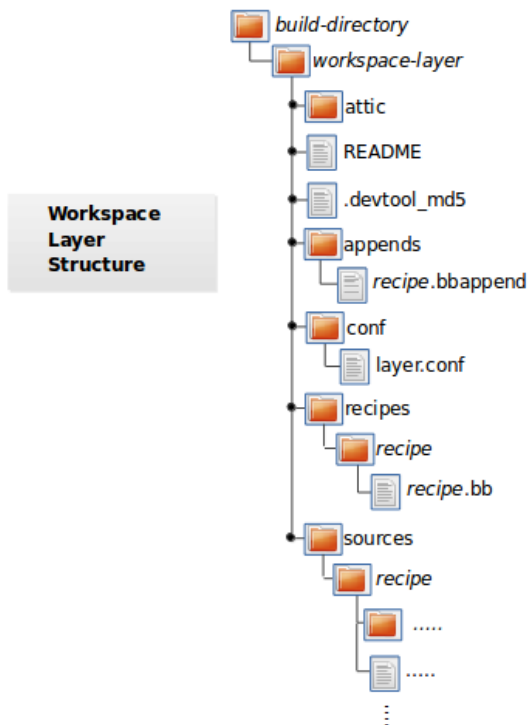
options:

|                                     |                                                                                                                                                         |
|-------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| -h, --help                          | show this help message and exit                                                                                                                         |
| --same-dir, -s                      | Build in same directory as source                                                                                                                       |
| --no-same-dir                       | Force build in a separate build directory                                                                                                               |
| --fetch URI, -f URI                 | Fetch the specified URI and extract it to create the source tree (deprecated - pass as positional argument instead)                                     |
| --fetch-dev                         | For npm, also fetch devDependencies                                                                                                                     |
| --version VERSION, -v VERSION       | Version to use within recipe (PV)                                                                                                                       |
| --no-git, -g                        | If fetching source, do not set up source tree as a git repository                                                                                       |
| --srcrev SRCREV, -S SRCREV          | Source revision to fetch if fetching from an SCM such as git (default latest)                                                                           |
| --autorev, -a                       | When fetching from a git repository, set SRCREV in the recipe to a floating revision instead of fixed                                                   |
| --srcbranch SRCBRANCH, -B SRCBRANCH | Branch in source repository if fetching from an SCM such as git (default master)                                                                        |
| --binary, -b                        | Treat the source tree as something that should be installed verbatim (no compilation, same directory structure). Useful with binary packages e.g. RPMs. |
| --also-native                       | Also add native variant (i.e. support building recipe for the build host as well as the target machine)                                                 |
| --src-subdir SUBDIR                 | Specify subdirectory within source tree to use                                                                                                          |
| --mirrors                           | Enable PREMIRRORS and MIRRORS for source tree fetching (disable by default).                                                                            |
| --provides PROVIDES, -p PROVIDES    | Specify an alias for the item provided by the recipe. E.g. virtual/libgl                                                                                |

## 28.2. The Workspace Layer Structure¶

devtool uses a "Workspace" layer in which to accomplish builds. This layer is not specific to any single devtool command but is rather a common working area used across the tool.

The following figure shows the workspace structure:



attic - A directory created if devtool believes it must preserve anything when you run "devtool reset". For example, if you run "devtool add", make changes to the recipe, and then run "devtool reset", devtool takes notice that the file has been changed and moves it into the attic should you still

want the recipe.

`README` - Provides information on what is in workspace layer and how to manage it.

`.devtool_md5` - A checksum file used by devtool.

`appends` - A directory that contains \*.bbappend files, which point to external source.

`conf` - A configuration directory that contains the layer.conf file.

`recipes` - A directory containing recipes. This directory contains a folder for each directory added whose name matches that of the added recipe. devtool places the *recipe.bb* file within that sub-directory.

`sources` - A directory containing a working copy of the source files used when building the recipe. This is the default directory used as the location of the source tree when you do not provide a source tree path. This directory contains a folder for each set of source files matched to a corresponding recipe.

### 28.3. Adding a New Recipe to the Workspace Layer¶

Use the `devtool add` command to add a new recipe to the workspace layer. The recipe you add should not exist - devtool creates it for you. The source files the recipe uses should exist in an external area.

The following example creates and adds a new recipe named `jackson` to a workspace layer the tool creates. The source code built by the recipes resides in `/home/user/sources/jackson`:

```
$ devtool add jackson /home/user/sources/jackson
```

If you add a recipe and the workspace layer does not exist, the command creates the layer and populates it as described in "[The Workspace Layer Structure](#)" section.

Running `devtool add` when the workspace layer exists causes the tool to add the recipe, append files, and source files into the existing workspace layer. The `.bbappend` file is created to point to the external source tree.

#### Note

If your recipe has runtime dependencies defined, you must be sure that these packages exist on the target hardware before attempting to run your application. If dependent packages (e.g. libraries) do not exist on the target, your application, when run, will fail to find those functions. For more information, see the "[Deploying Your Software on the Target Machine](#)" section.

By default, `devtool add` uses the latest revision (i.e. master) when unpacking files from a remote URI. In some cases, you might want to specify a source revision by branch, tag, or commit hash. You can specify these options when using the `devtool add` command:

- To specify a source branch, use the `--srcbranch` option:

```
$ devtool add --srcbranch zeus jackson /home/user/sources/jackson
```

In the previous example, you are checking out the zeus branch.

- To specify a specific tag or commit hash, use the `--srcrev` option:

```
$ devtool add --srcrev yocto-3.0.1 jackson /home/user/sources/jackson
$ devtool add --srcrev some_commit_hash /home/user/sources/jackson
```

The previous examples check out the yocto-3.0.1 tag and the commit associated with the *some\_commit\_hash* hash.

#### Note

If you prefer to use the latest revision every time the recipe is built, use the options `--autorev` or `-a`.

### 28.4. Extracting the Source for an Existing Recipe¶

Use the `devtool extract` command to extract the source for an existing recipe. When you use this command, you must supply the root name of the recipe (i.e. no version, paths, or extensions), and you must supply the directory to which you want the source extracted.

Additional command options let you control the name of a development branch into which you can checkout the source and whether or not to keep a temporary directory, which is useful for debugging.

### 28.5. Synchronizing a Recipe's Extracted Source Tree¶

Use the `devtool sync` command to synchronize a previously extracted source tree for an existing recipe. When you use this command, you must supply the root name of the recipe (i.e. no version, paths, or extensions), and you must supply the directory to which you want the source extracted.



Additional command options let you control the name of a development branch into which you can checkout the source and whether or not to keep a temporary directory, which is useful for debugging.

## 28.6. Modifying an Existing Recipe¶

Use the `devtool modify` command to begin modifying the source of an existing recipe. This command is very similar to the `add` command except that it does not physically create the recipe in the workspace layer because the recipe already exists in another layer.

The `devtool modify` command extracts the source for a recipe, sets it up as a Git repository if the source had not already been fetched from Git, checks out a branch for development, and applies any patches from the recipe as commits on top. You can use the following command to checkout the source files:

```
$ devtool modify recipe
```

Using the above command form, `devtool` uses the existing recipe's `SRC_URI` statement to locate the upstream source, extracts the source into the default sources location in the workspace. The default development branch used is "devtool".

## 28.7. Edit an Existing Recipe¶

Use the `devtool edit-recipe` command to run the default editor, which is identified using the `EDITOR` variable, on the specified recipe.

When you use the `devtool edit-recipe` command, you must supply the root name of the recipe (i.e. no version, paths, or extensions). Also, the recipe file itself must reside in the workspace as a result of the `devtool add` or `devtool upgrade` commands. However, you can override that requirement by using the `-a` or `--any-recipe` option. Using either of these options allows you to edit any recipe regardless of its location.

## 28.8. Updating a Recipe¶

Use the `devtool update-recipe` command to update your recipe with patches that reflect changes you make to the source files. For example, if you know you are going to work on some code, you could first use the `devtool modify` command to extract the code and set up the workspace. After which, you could modify, compile, and test the code.

When you are satisfied with the results and you have committed your changes to the Git repository, you can then run the `devtool update-recipe` to create the patches and update the recipe:

```
$ devtool update-recipe recipe
```

If you run the `devtool update-recipe` without committing your changes, the command ignores the changes.

Often, you might want to apply customizations made to your software in your own layer rather than apply them to the original recipe. If so, you can use the `-a` or `--append` option with the `devtool update-recipe` command. These options allow you to specify the layer into which to write an append file:

```
$ devtool update-recipe recipe -a base-layer-directory
```

The `*.bbappend` file is created at the appropriate path within the specified layer directory, which may or may not be in your `bblayers.conf` file. If an append file already exists, the command updates it appropriately.

## 28.9. Checking on the Upgrade Status of a Recipe¶

Upstream recipes change over time. Consequently, you might find that you need to determine if you can upgrade a recipe to a newer version.

To check on the upgrade status of a recipe, use the `devtool check-upgrade-status` command. The command displays a table of your current recipe versions, the latest upstream versions, the email address of the recipe's maintainer, and any additional information such as commit hash strings and reasons you might not be able to upgrade a particular recipe.

### NOTES:

- For the `oe-core` layer, recipe maintainers come from the `maintainers.inc` file.
- If the recipe is using the `Git fetcher` rather than a tarball, the commit hash points to the commit that matches the recipe's latest version tag.

As with all `devtool` commands, you can get help on the individual command:

```
$ devtool check-upgrade-status -h
NOTE: Starting bitbake server...
usage: devtool check-upgrade-status [-h] [--all] [recipe [recipe ...]]

Prints a table of recipes together with versions currently provided by
recipes, and latest upstream versions, when there is a later version available

arguments:
 recipe Name of the recipe to report (omit to report upgrade info for
 all recipes)

options:
```

```
-h, --help show this help message and exit
--all, -a Show all recipes, not just recipes needing upgrade
```

Unless you provide a specific recipe name on the command line, the command checks all recipes in all configured layers.

Following is a partial example table that reports on all the recipes. Notice the reported reason for not upgrading the `base-passwd` recipe. In this example, while a new version is available upstream, you do not want to use it because the dependency on `cdebconf` is not easily satisfied.

### Note

When a reason for not upgrading displays, the reason is usually written into the recipe using the `RECIPE_NO_UPDATE_REASON` variable. See the `base-passwd.bb` recipe for an example.

```
$ devtool check-upgrade-status
...
NOTE: acpid 2.0.30 2.0.31
Ross Burton <ross.burton@intel.com>
NOTE: u-boot-fw-utils 2018.11 2019.01
Marek Vasut <marek.vasut@gmail.com>
d3689267f92c5956e09cc7d1baa4700141662bff
NOTE: u-boot-tools 2018.11 2019.01
Marek Vasut <marek.vasut@gmail.com>
d3689267f92c5956e09cc7d1baa4700141662bff
.
.
NOTE: base-passwd 3.5.29 3.5.45
Anuj Mittal <anuj.mittal@intel.com> cannot be updated due to: Version
3.5.38 requires cdebconf for update-passwd utility
NOTE: busybox 1.29.2 1.30.0
Andrej Valek <andrej.valek@siemens.com>
NOTE: dbus-test 1.12.10 1.12.12
Chen Qi <Qi.Chen@windriver.com>
```

## 28.10. Upgrading a Recipe¶

As software matures, upstream recipes are upgraded to newer versions. As a developer, you need to keep your local recipes up-to-date with the upstream version releases. Several methods exist by which you can upgrade recipes. You can read about them in the "[Upgrading Recipes](#)" section of the Yocto Project Development Tasks Manual. This section overviews the `devtool upgrade` command.

### Note

Before you upgrade a recipe, you can check on its upgrade status. See the "[Checking on the Upgrade Status of a Recipe](#)" for more information.

The `devtool upgrade` command upgrades an existing recipe to a more recent version of the recipe upstream. The command puts the upgraded recipe file along with any associated files into a "workspace" and, if necessary, extracts the source tree to a specified location. During the upgrade, patches associated with the recipe are rebased or added as needed.

When you use the `devtool upgrade` command, you must supply the root name of the recipe (i.e. no version, paths, or extensions), and you must supply the directory to which you want the source extracted. Additional command options let you control things such as the version number to which you want to upgrade (i.e. the `PV`), the source revision to which you want to upgrade (i.e. the `SRCREV`), whether or not to apply patches, and so forth.

You can read more on the `devtool upgrade` workflow in the "[Use devtool upgrade to Create a Version of the Recipe that Supports a Newer Version of the Software](#)" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual. You can also see an example of how to use `devtool upgrade` in the "[Using devtool upgrade](#)" section in the Yocto Project Development Tasks Manual.

## 28.11. Resetting a Recipe¶

Use the `devtool reset` command to remove a recipe and its configuration (e.g. the corresponding `.bbappend` file) from the workspace layer. Realize that this command deletes the recipe and the append file. The command does not physically move them for you. Consequently, you must be sure to physically relocate your updated recipe and the append file outside of the workspace layer before running the `devtool reset` command.

If the `devtool reset` command detects that the recipe or the append files have been modified, the command preserves the modified files in a separate "attic" subdirectory under the workspace layer.

Here is an example that resets the workspace directory that contains the `mtr` recipe:

```
$ devtool reset mtr
NOTE: Cleaning sysroot for recipe mtr...
NOTE: Leaving source tree /home/scottrif/poky/build/workspace/sources/mtr as-is; if you no
longer need it then please delete it manually
$
```

## 28.12. Building Your Recipe¶

Use the `devtool build` command to build your recipe. The `devtool build` command is equivalent to the `bitbake -c populate_sysroot` command.

When you use the `devtool build` command, you must supply the root name of the recipe (i.e. do not provide versions, paths, or extensions). You can use either the `-s` or the `--disable-parallel-make` options to disable parallel makes during the build. Here is an example:

```
$ devtool build recipe
```

### 28.13. Building Your Image¶

Use the `devtool build-image` command to build an image, extending it to include packages from recipes in the workspace. Using this command is useful when you want an image that ready for immediate deployment onto a device for testing. For proper integration into a final image, you need to edit your custom image recipe appropriately.

When you use the `devtool build-image` command, you must supply the name of the image. This command has no command line options:

```
$ devtool build-image image
```

### 28.14. Deploying Your Software on the Target Machine¶

Use the `devtool deploy-target` command to deploy the recipe's build output to the live target machine:

```
$ devtool deploy-target recipe target
```

The *target* is the address of the target machine, which must be running an SSH server (i.e. `user@hostname[:destdir]`).

This command deploys all files installed during the `do_install` task. Furthermore, you do not need to have package management enabled within the target machine. If you do, the package manager is bypassed.

#### Notes

The `deploy-target` functionality is for development only. You should never use it to update an image that will be used in production.

Some conditions exist that could prevent a deployed application from behaving as expected. When both of the following conditions exist, your application has the potential to not behave correctly when run on the target:

- You are deploying a new application to the target and the recipe you used to build the application had correctly defined runtime dependencies.
- The target does not physically have the packages on which the application depends installed.

If both of these conditions exist, your application will not behave as expected. The reason for this misbehavior is because the `devtool deploy-target` command does not deploy the packages (e.g. libraries) on which your new application depends. The assumption is that the packages are already on the target. Consequently, when a runtime call is made in the application for a dependent function (e.g. a library call), the function cannot be found.

To be sure you have all the dependencies local to the target, you need to be sure that the packages are pre-deployed (installed) on the target before attempting to run your application.

### 28.15. Removing Your Software from the Target Machine¶

Use the `devtool undeploy-target` command to remove deployed build output from the target machine. For the `devtool undeploy-target` command to work, you must have previously used the `devtool deploy-target` command.

```
$ devtool undeploy-target recipe target
```

The *target* is the address of the target machine, which must be running an SSH server (i.e. `user@hostname`).

### 28.16. Creating the Workspace Layer in an Alternative Location¶

Use the `devtool create-workspace` command to create a new workspace layer in your [Build Directory](#). When you create a new workspace layer, it is populated with the `README` file and the `conf` directory only.

The following example creates a new workspace layer in your current working and by default names the workspace layer "workspace":

```
$ devtool create-workspace
```

You can create a workspace layer anywhere by supplying a pathname with the command. The following command creates a new workspace layer named "new-workspace":

```
$ devtool create-workspace /home/scottrif/new-workspace
```

## 28.17. Get the Status of the Recipes in Your Workspace¶

Use the `devtool status` command to list the recipes currently in your workspace. Information includes the paths to their respective external source trees.

The `devtool status` command has no command-line options:

```
$ devtool status
```

Following is sample output after using `devtool add` to create and add the `mtr_0.86.bb` recipe to the `workspace` directory:

```
$ devtool status
mtr: /home/scottrif/poky/build/workspace/sources/mtr (/home/scottrif/poky/build/workspace/recipes/mtr/mtr_0.86.bb)
$
```

## 28.18. Search for Available Target Recipes¶

Use the `devtool search` command to search for available target recipes. The command matches the recipe name, package name, description, and installed files. The command displays the recipe name as a result of a match.

When you use the `devtool search` command, you must supply a *keyword*. The command uses the *keyword* when searching for a match.

## Chapter 29. OpenEmbedded Kickstart (.wks) Reference¶

### [29.1. Introduction](#)

### [29.2. Command: part or partition](#)

### [29.3. Command: bootloader](#)

## 29.1. Introduction¶

The current Wic implementation supports only the basic kickstart partitioning commands: `partition` (or `part` for short) and `bootloader`.

### Note

Future updates will implement more commands and options. If you use anything that is not specifically supported, results can be unpredictable.

This chapter provides a reference on the available kickstart commands. The information lists the commands, their syntax, and meanings. Kickstart commands are based on the Fedora kickstart versions but with modifications to reflect Wic capabilities. You can see the original documentation for those commands at the following link:

<http://pykickstart.readthedocs.io/en/latest/kickstart-docs.html>

## 29.2. Command: part or partition¶

Either of these commands creates a partition on the system and uses the following syntax:

```
part [mntpoint]
partition [mntpoint]
```

If you do not provide *mntpoint*, Wic creates a partition but does not mount it.

The *mntpoint* is where the partition is mounted and must be in one of the following forms:

- */path*: For example, `/`, `/usr`, or `/home`
- `swap`: The created partition is used as swap space

Specifying a *mntpoint* causes the partition to automatically be mounted. Wic achieves this by adding entries to the filesystem table (`fstab`) during image generation. In order for Wic to generate a valid `fstab`, you must also provide one of the `--ondrive`, `--ondisk`, or `--use-uuid` partition options as part of the command.

### Note

The mount program must understand the PARTUUID syntax you use with `--use-uuid` and non-root *mountpoint*, including `swap`. The busybox versions of these application are currently excluded.

Here is an example that uses `/` as the *mountpoint*. The command uses `--ondisk` to force the partition onto the `sdb` disk:

```
part / --source rootfs --ondisk sdb --fstype=ext3 --label platform --align 1024
```

Here is a list that describes other supported options you can use with the `part` and `partition` commands:

- **--size:** The minimum partition size in MBytes. Specify an integer value such as 500. Do not append the number with "MB". You do not need this option if you use `--source`.
- **--fixed-size:** The exact partition size in MBytes. You cannot specify with `--size`. An error occurs when assembling the disk image if the partition data is larger than `--fixed-size`.
- **--source:** This option is a Wic-specific option that names the source of the data that populates the partition. The most common value for this option is "rootfs", but you can use any value that maps to a valid source plugin. For information on the source plugins, see the "[Using the Wic Plugins Interface](#)" section in the Yocto Project Development Tasks Manual.

If you use `--source rootfs`, Wic creates a partition as large as needed and fills it with the contents of the root filesystem pointed to by the `-r` command-line option or the equivalent rootfs derived from the `-e` command-line option. The filesystem type used to create the partition is driven by the value of the `--fstype` option specified for the partition. See the entry on `--fstype` that follows for more information.

If you use `--source plugin-name`, Wic creates a partition as large as needed and fills it with the contents of the partition that is generated by the specified plugin name using the data pointed to by the `-r` command-line option or the equivalent rootfs derived from the `-e` command-line option. Exactly what those contents are and filesystem type used are dependent on the given plugin implementation.

If you do not use the `--source` option, the `wic` command creates an empty partition. Consequently, you must use the `--size` option to specify the size of the empty partition.

- **--ondisk or --ondrive:** Forces the partition to be created on a particular disk.
- **--fstype:** Sets the file system type for the partition. Valid values are:
  - `ext4`
  - `ext3`
  - `ext2`
  - `btrfs`
  - `squashfs`
  - `swap`
- **--fsoptions:** Specifies a free-form string of options to be used when mounting the filesystem. This string is copied into the `/etc/fstab` file of the installed system and should be enclosed in quotes. If not specified, the default string is "defaults".
- **--label label:** Specifies the label to give to the filesystem to be made on the partition. If the given label is already in use by another filesystem, a new label is created for the partition.
- **--active:** Marks the partition as active.
- **--align (in KBytes):** This option is a Wic-specific option that says to start partitions on boundaries given x KBytes.
- **--no-table:** This option is a Wic-specific option. Using the option reserves space for the partition and causes it to become populated. However, the partition is not added to the partition table.
- **--exclude-path:** This option is a Wic-specific option that excludes the given relative path from the resulting image. This option is only effective with the rootfs source plugin.
- **--extra-space:** This option is a Wic-specific option that adds extra space after the space filled by the content of the partition. The final size can exceed the size specified by the `--size` option. The default value is 10 Mbytes.
- **--overhead-factor:** This option is a Wic-specific option that multiplies the size of the partition by the option's value. You must supply a value greater than or equal to "1". The default value is "1.3".
- **--part-name:** This option is a Wic-specific option that specifies a name for GPT partitions.
- **--part-type:** This option is a Wic-specific option that specifies the partition type globally unique identifier (GUID) for GPT partitions. You can find the list of partition type GUIDs at [http://en.wikipedia.org/wiki/GUID\\_Partition\\_Table#Partition\\_type\\_GUIDs](http://en.wikipedia.org/wiki/GUID_Partition_Table#Partition_type_GUIDs).
- **--use-uuid:** This option is a Wic-specific option that causes Wic to generate a random GUID for the partition. The generated identifier is used in the bootloader configuration to specify the root partition.
- **--uuid:** This option is a Wic-specific option that specifies the partition UUID.
- **--fsuuid:** This option is a Wic-specific option that specifies the filesystem UUID. You can generate or modify `WKS_FILE` with this option if a preconfigured filesystem UUID is added to the kernel command line in the bootloader configuration before you run Wic.
- **--system-id:** This option is a Wic-specific option that specifies the partition system ID, which is a one byte long, hexadecimal parameter with or without the 0x prefix.
- **--mkfs-extraopts:** This option specifies additional options to pass to the `mkfs` utility. Some default options for certain filesystems do not take effect. See Wic's help on kickstart (i.e. `wic help kickstart`).

## 29.3. Command: `bootloader`

This command specifies how the bootloader should be configured and supports the following options:

### Note

Bootloader functionality and boot partitions are implemented by the various `--source` plugins that implement bootloader functionality. The `bootloader` command essentially provides a means of modifying

bootloader configuration.

- **--timeout:** Specifies the number of seconds before the bootloader times out and boots the default option.
- **--append:** Specifies kernel parameters. These parameters will be added to the syslinux `APPEND` or `grub` kernel command line.
- **--configfile:** Specifies a user-defined configuration file for the bootloader. You can provide a full pathname for the file or a file that exists in the `canned-wks` folder. This option overrides all other bootloader options.

## Chapter 30. QA Error and Warning Messages¶

[30.1. Introduction](#)

[30.2. Errors and Warnings](#)

[30.3. Configuring and Disabling QA Checks](#)

### 30.1. Introduction¶

When building a recipe, the OpenEmbedded build system performs various QA checks on the output to ensure that common issues are detected and reported. Sometimes when you create a new recipe to build new software, it will build with no problems. When this is not the case, or when you have QA issues building any software, it could take a little time to resolve them.

While it is tempting to ignore a QA message or even to disable QA checks, it is best to try and resolve any reported QA issues. This chapter provides a list of the QA messages and brief explanations of the issues you could encounter so that you can properly resolve problems.

The next section provides a list of all QA error and warning messages based on a default configuration. Each entry provides the message or error form along with an explanation.

### Notes

- At the end of each message, the name of the associated QA test (as listed in the `"insane.bbclass"` section) appears within square brackets.
- As mentioned, this list of error and warning messages is for QA checks only. The list does not cover all possible build errors or warnings you could encounter.
- Because some QA checks are disabled by default, this list does not include all possible QA check errors and warnings.

### 30.2. Errors and Warnings¶

- `<package>: <path> is using libexec please relocate to <libexecdir> [libexec]`

The specified package contains files in `/usr/libexec` when the distro configuration uses a different path for `<libexecdir>`. By default, `<libexecdir>` is `$prefix/libexec`. However, this default can be changed (e.g. `${libdir}`).

- `package <package> contains bad RPATH <rpath> in file <file> [rpaths]`

The specified binary produced by the recipe contains dynamic library load paths (rpaths) that contain build system paths such as `TMPDIR`, which are incorrect for the target and could potentially be a security issue. Check for bad `-rpath` options being passed to the linker in your `do_compile` log. Depending on the build system used by the software being built, there might be a configure option to disable rpath usage completely within the build of the software.

- `<package>: <file> contains probably-redundant RPATH <rpath> [useless-rpaths]`

The specified binary produced by the recipe contains dynamic library load paths (rpaths) that on a standard system are searched by default by the linker (e.g. `/lib` and `/usr/lib`). While these paths will not cause any breakage, they do waste space and are unnecessary. Depending on the build system used by the software being built, there might be a configure option to disable rpath usage completely within the build of the software.

- `<package> requires <files>, but no providers in its RDEPENDS [file-rdeps]`

A file-level dependency has been identified from the specified package on the specified files, but there is no explicit corresponding entry in `RDEPENDS`. If particular files are required at runtime then `RDEPENDS` should be declared in the recipe to ensure the packages providing them are built.

- `<package1> rdepends on <package2>, but it isn't a build dependency? [build-deps]`

A runtime dependency exists between the two specified packages, but there is nothing explicit within the recipe to enable the OpenEmbedded build system to ensure that dependency is satisfied. This condition is usually triggered by an [RDEPENDS](#) value being added at the packaging stage rather than up front, which is usually automatic based on the contents of the package. In most cases, you should change the recipe to add an explicit [RDEPENDS](#) for the dependency.

- `non -dev/-dbg/nativesdk- package contains symlink .so: <packagename> path '<path>' [dev-so]`

Symlink `.so` files are for development only, and should therefore go into the `-dev` package. This situation might occur if you add `*.so*` rather than `*.so.*` to a non-dev package. Change [FILES](#) (and possibly [PACKAGES](#)) such that the specified `.so` file goes into an appropriate `-dev` package.

- `non -staticdev package contains static .a library: <packagename> path '<path>' [staticdev]`

Static `.a` library files should go into a `-staticdev` package. Change [FILES](#) (and possibly [PACKAGES](#)) such that the specified `.a` file goes into an appropriate `-staticdev` package.

- `<packagename>: found library in wrong location [libdir]`

The specified file may have been installed into an incorrect (possibly hardcoded) installation path. For example, this test will catch recipes that install `/lib/bar.so` when `${base_libdir}` is `"lib32"`. Another example is when recipes install `/usr/lib64/foo.so` when `${libdir}` is `"usr/lib"`. False positives occasionally exist. For these cases add `"libdir"` to [INSANE\\_SKIP](#) for the package.

- `non debug package contains .debug directory: <packagename> path <path> [debug-files]`

The specified package contains a `.debug` directory, which should not appear in anything but the `-dbg` package. This situation might occur if you add a path which contains a `.debug` directory and do not explicitly add the `.debug` directory to the `-dbg` package. If this is the case, add the `.debug` directory explicitly to `FILES_${PN}-dbg`. See [FILES](#) for additional information on [FILES](#).

- `Architecture did not match (<machine_arch> to <file_arch>) on <file> [arch]`

By default, the OpenEmbedded build system checks the Executable and Linkable Format (ELF) type, bit size, and endianness of any binaries to ensure they match the target architecture. This test fails if any binaries do not match the type since there would be an incompatibility. The test could indicate that the wrong compiler or compiler options have been used. Sometimes software, like bootloaders, might need to bypass this check. If the file you receive the error for is firmware that is not intended to be executed within the target operating system or is intended to run on a separate processor within the device, you can add `"arch"` to [INSANE\\_SKIP](#) for the package. Another option is to check the [do\\_compile](#) log and verify that the compiler options being used are correct.

- `Bit size did not match (<machine_bits> to <file_bits>) <recipe> on <file> [arch]`

By default, the OpenEmbedded build system checks the Executable and Linkable Format (ELF) type, bit size, and endianness of any binaries to ensure they match the target architecture. This test fails if any binaries do not match the type since there would be an incompatibility. The test could indicate that the wrong compiler or compiler options have been used. Sometimes software, like bootloaders, might need to bypass this check. If the file you receive the error for is firmware that is not intended to be executed within the target operating system or is intended to run on a separate processor within the device, you can add `"arch"` to [INSANE\\_SKIP](#) for the package. Another option is to check the [do\\_compile](#) log and verify that the compiler options being used are correct.

- `Endianness did not match (<machine_endianness> to <file_endianness>) on <file> [arch]`

By default, the OpenEmbedded build system checks the Executable and Linkable Format (ELF) type, bit size, and endianness of any binaries to ensure they match the target architecture. This test fails if any binaries do not match the type since there would be an incompatibility. The test could indicate that the wrong compiler or compiler options have been used. Sometimes software, like bootloaders, might need to bypass this check. If the file you receive the error for is firmware that is not intended to be executed within the target operating system or is intended to run on a separate processor within the device, you can add `"arch"` to [INSANE\\_SKIP](#) for the package. Another option is to check the [do\\_compile](#) log and verify that the compiler options being used are correct.

- `ELF binary '<file>' has relocations in .text [textrel]`

The specified ELF binary contains relocations in its `.text` sections. This situation can result in a performance impact at runtime.

Typically, the way to solve this performance issue is to add `"-fPIC"` or `"-fpic"` to the compiler command-line options. For example, given software that reads [CFLAGS](#) when you build it, you could add the following to your recipe:

```
CFLAGS_append = " -fPIC "
```

For more information on text relocations at runtime, see <http://www.akkadia.org/drepper/textrelocs.html>.

- `No GNU_HASH in the elf binary: '<file>' [ldflags]`

This indicates that binaries produced when building the recipe have not been linked with the [LDFLAGS](#) options provided by the build system. Check to be sure that the [LDFLAGS](#) variable is being passed to the linker command. A common workaround for this

situation is to pass in `LDFLAGS` using `TARGET_CC_ARCH` within the recipe as follows:

```
TARGET_CC_ARCH += "${LDFLAGS}"
```

- Package `<packagename>` contains Xorg driver (`<driver>`) but no `xorg-abi-` dependencies [`xorg-driver-abi`]

The specified package contains an Xorg driver, but does not have a corresponding ABI package dependency. The `xserver-xorg` recipe provides driver ABI names. All drivers should depend on the ABI versions that they have been built against. Driver recipes that include `xorg-driver-input.inc` or `xorg-driver-video.inc` will automatically get these versions. Consequently, you should only need to explicitly add dependencies to binary driver recipes.

- The `/usr/share/info/dir` file is not meant to be shipped in a particular package. [`infodir`]

The `/usr/share/info/dir` should not be packaged. Add the following line to your `do_install` task or to your `do_install_append` within the recipe as follows:

```
rm ${D}${infodir}/dir
```

- Symlink `<path>` in `<packagename>` points to `TMPDIR` [`symlink-to-sysroot`]

The specified symlink points into `TMPDIR` on the host. Such symlinks will work on the host. However, they are clearly invalid when running on the target. You should either correct the symlink to use a relative path or remove the symlink.

- `<file>` failed sanity test (`workdir`) in path `<path>` [`la`]

The specified `.la` file contains `TMPDIR` paths. Any `.la` file containing these paths is incorrect since `libtool` adds the correct sysroot prefix when using the files automatically itself.

- `<file>` failed sanity test (`tmpdir`) in path `<path>` [`pkgconfig`]

The specified `.pc` file contains `TMPDIR`/`WORKDIR` paths. Any `.pc` file containing these paths is incorrect since `pkg-config` itself adds the correct sysroot prefix when the files are accessed.

- `<packagename>` `rdepends` on `<debug_packagename>` [`debug-deps`]

A dependency exists between the specified non-dbg package (i.e. a package whose name does not end in `-dbg`) and a package that is a `dbg` package. The `dbg` packages contain debug symbols and are brought in using several different methods:

- Using the `dbg-pkgs` `IMAGE_FEATURES` value.
- Using `IMAGE_INSTALL`.
- As a dependency of another `dbg` package that was brought in using one of the above methods.

The dependency might have been automatically added because the `dbg` package erroneously contains files that it should not contain (e.g. a non-symlink `.so` file) or it might have been added manually (e.g. by adding to `RDEPENDS`).

- `<packagename>` `rdepends` on `<dev_packagename>` [`dev-deps`]

A dependency exists between the specified non-dev package (a package whose name does not end in `-dev`) and a package that is a `dev` package. The `dev` packages contain development headers and are usually brought in using several different methods:

- Using the `dev-pkgs` `IMAGE_FEATURES` value.
- Using `IMAGE_INSTALL`.
- As a dependency of another `dev` package that was brought in using one of the above methods.

The dependency might have been automatically added (because the `dev` package erroneously contains files that it should not have (e.g. a non-symlink `.so` file) or it might have been added manually (e.g. by adding to `RDEPENDS`).

- `<var>_<packagename>` is invalid: `<comparison>` (`<value>`) only comparisons `<`, `=`, `>`, `<=`, and `>=` are allowed [`dep-cmp`]

If you are adding a versioned dependency relationship to one of the dependency variables (`RDEPENDS`, `RRECOMMENDS`, `RSUGGESTS`, `RPROVIDES`, `RREPLACES`, or `RCONFLICTS`), you must only use the named comparison operators. Change the versioned dependency values you are adding to match those listed in the message.

- `<recipe>`: The compile log indicates that host include and/or library paths were used. Please check the log '`<logfile>`' for more information. [`compile-host-path`]

The log for the `do_compile` task indicates that paths on the host were searched for files, which is not appropriate when cross-compiling. Look for "is unsafe for cross-compilation" or "CROSS COMPILE Badness" in the specified log file.

- `<recipe>`: The install log indicates that host include and/or library paths were used. Please check the log '`<logfile>`' for more information. [`install-host-path`]



The log for the [do\\_install](#) task indicates that paths on the host were searched for files, which is not appropriate when cross-compiling. Look for "is unsafe for cross-compilation" or "CROSS COMPILE Badness" in the specified log file.

- This autoconf log indicates errors, it looked at host include and/or library paths while determining system capabilities. Rerun configure task after fixing this. The path was '<path>'

The log for the [do\\_configure](#) task indicates that paths on the host were searched for files, which is not appropriate when cross-compiling. Look for "is unsafe for cross-compilation" or "CROSS COMPILE Badness" in the specified log file.

- <packagename> doesn't match the [a-z0-9.-]+ regex [pkgname]

The convention within the OpenEmbedded build system (sometimes enforced by the package manager itself) is to require that package names are all lower case and to allow a restricted set of characters. If your recipe name does not match this, or you add packages to [PACKAGES](#) that do not conform to the convention, then you will receive this error. Rename your recipe. Or, if you have added a non-conforming package name to [PACKAGES](#), change the package name appropriately.

- <recipe>: configure was passed unrecognized options: <options> [unknown-configure-option]

The configure script is reporting that the specified options are unrecognized. This situation could be because the options were previously valid but have been removed from the configure script. Or, there was a mistake when the options were added and there is another option that should be used instead. If you are unsure, consult the upstream build documentation, the `./configure --help` output, and the upstream change log or release notes. Once you have worked out what the appropriate change is, you can update [EXTRA\\_OECONF](#), [PACKAGECONFIG\\_CONFGARGS](#), or the individual [PACKAGECONFIG](#) option values accordingly.

- Recipe <recipefile> has PN of "<recipe>" which is in [OVERRIDES](#), this can result in unexpected behavior. [pn-overrides]

The specified recipe has a name ([PN](#)) value that appears in [OVERRIDES](#). If a recipe is named such that its PN value matches something already in [OVERRIDES](#) (e.g. PN happens to be the same as [MACHINE](#) or [DISTRO](#)), it can have unexpected consequences. For example, assignments such as `FILES_${PN} = "xyz"` effectively turn into `FILES = "xyz"`. Rename your recipe (or if PN is being set explicitly, change the PN value) so that the conflict does not occur. See [FILES](#) for additional information.

- <recipefile>: Variable <variable> is set as not being package specific, please fix this. [pkgvarcheck]

Certain variables ([RDEPENDS](#), [RRECOMMENDS](#), [RSUGGESTS](#), [RCONFLICTS](#), [RPROVIDES](#), [RREPLACES](#), [FILES](#), `pkg_preinst`, `pkg_postinst`, `pkg_preun`, `pkg_postrm`, and [ALLOW\\_EMPTY](#)) should always be set specific to a package (i.e. they should be set with a package name override such as `RDEPENDS_${PN} = "value"` rather than `RDEPENDS = "value"`). If you receive this error, correct any assignments to these variables within your recipe.

- File '<file>' from <recipe> was already stripped, this will prevent future debugging! [already-stripped]

Produced binaries have already been stripped prior to the build system extracting debug symbols. It is common for upstream software projects to default to stripping debug symbols for output binaries. In order for debugging to work on the target using `-dbg` packages, this stripping must be disabled.

Depending on the build system used by the software being built, disabling this stripping could be as easy as specifying an additional configure option. If not, disabling stripping might involve patching the build scripts. In the latter case, look for references to "strip" or "STRIP", or the "-s" or "-S" command-line options being specified on the linker command line (possibly through the compiler command line if preceded with "-Wl,").

## Note

Disabling stripping here does not mean that the final packaged binaries will be unstripped. Once the OpenEmbedded build system splits out debug symbols to the `-dbg` package, it will then strip the symbols from the binaries.

- <packagename> is listed in [PACKAGES](#) multiple times, this leads to packaging errors. [packages-list]

Package names must appear only once in the [PACKAGES](#) variable. You might receive this error if you are attempting to add a package to [PACKAGES](#) that is already in the variable's value.

- [FILES](#) variable for package <packagename> contains '/' which is invalid. Attempting to fix this but you should correct the metadata. [files-invalid]

The string "/" is invalid in a Unix path. Correct all occurrences where this string appears in a [FILES](#) variable so that there is only a single "/".

- <recipe>: Files/directories were installed but not shipped in any package [installed-vs-shipped]

Files have been installed within the [do\\_install](#) task but have not been included in any package by way of the [FILES](#) variable. Files that do not appear in any package cannot be present in an image later on in the build process. You need to do one of

the following:

- Add the files to `FILES` for the package you want them to appear in (e.g. `FILES_${PN}` for the main package).
  - Delete the files at the end of the `do_install` task if the files are not needed in any package.
- `<oldpackage>-<oldpkgversion>` was registered as shlib provider for `<library>`, changing it to `<newpackage>-<newpkgversion>` because it was built later

This message means that both `<oldpackage>` and `<newpackage>` provide the specified shared library. You can expect this message when a recipe has been renamed. However, if that is not the case, the message might indicate that a private version of a library is being erroneously picked up as the provider for a common library. If that is the case, you should add the library's `.SO` file name to `PRIVATE_LIBS` in the recipe that provides the private version of the library.

### 30.3. Configuring and Disabling QA Checks¶

You can configure the QA checks globally so that specific check failures either raise a warning or an error message, using the `WARN_QA` and `ERROR_QA` variables, respectively. You can also disable checks within a particular recipe using `INSANE_SKIP`. For information on how to work with the QA checks, see the "[insane.bbclass](#)" section.

#### Tip

Please keep in mind that the QA checks exist in order to detect real or potential problems in the packaged output. So exercise caution when disabling these checks.

## Chapter 31. Images¶

The OpenEmbedded build system provides several example images to satisfy different needs. When you issue the `bitbake` command you provide a "top-level" recipe that essentially begins the build for the type of image you want.

#### Note

Building an image without GNU General Public License Version 3 (GPLv3), GNU Lesser General Public License Version 3 (LGPLv3), and the GNU Affero General Public License Version 3 (AGPL-3.0) components is only supported for minimal and base images. Furthermore, if you are going to build an image using non-GPLv3 and similarly licensed components, you must make the following changes in the `local.conf` file before using the BitBake command to build the minimal or base image:

1. Comment out the `EXTRA_IMAGE_FEATURES` line
2. Set `INCOMPATIBLE_LICENSE = "GPL-3.0 LGPL-3.0 AGPL-3.0"`

From within the `poky` Git repository, you can use the following command to display the list of directories within the [Source Directory](#) that contain image recipe files:

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

Following is a list of supported recipes:

- `build-appliance-image`: An example virtual machine that contains all the pieces required to run builds using the build system as well as the build system itself. You can boot and run the image using either the [VMware Player](#) or [VMware Workstation](#). For more information on this image, see the [Build Appliance](#) page on the Yocto Project website.
- `core-image-base`: A console-only image that fully supports the target device hardware.
- `core-image-clutter`: An image with support for the Open GL-based toolkit Clutter, which enables development of rich and animated graphical user interfaces.
- `core-image-full-cmdline`: A console-only image with more full-featured Linux system functionality installed.
- `core-image-lsb`: An image that conforms to the Linux Standard Base (LSB) specification. This image requires a distribution configuration that enables LSB compliance (e.g. `poky-lsb`). If you build `core-image-lsb` without that configuration, the image will not be LSB-compliant.
- `core-image-lsb-dev`: A `core-image-lsb` image that is suitable for development work using the host. The image includes headers and libraries you can use in a host development environment. This image requires a distribution configuration that enables LSB compliance (e.g. `poky-lsb`). If you build `core-image-lsb-dev` without that configuration, the image will not be LSB-compliant.
- `core-image-lsb-sdk`: A `core-image-lsb` that includes everything in the cross-toolchain but also includes development headers and libraries to form a complete standalone SDK. This image requires a distribution configuration that enables LSB compliance (e.g. `poky-lsb`). If you build `core-image-lsb-sdk` without that configuration, the image will not be LSB-compliant. This image is suitable for development using the target.
- `core-image-minimal`: A small image just capable of allowing a device to boot.
- `core-image-minimal-dev`: A `core-image-minimal` image suitable for development work using the host. The image includes headers and libraries you can use in a host development environment.
- `core-image-minimal-initramfs`: A `core-image-minimal` image that has the Minimal RAM-based Initial Root Filesystem (initramfs) as part of the kernel, which allows the system to find the first "init" program more efficiently. See the

`PACKAGE_INSTALL` variable for additional information helpful when working with initramfs images.

- `core-image-minimal-mtdutils`: A `core-image-minimal` image that has support for the Minimal MTD Utilities, which let the user interact with the MTD subsystem in the kernel to perform operations on flash devices.
- `core-image-rt`: A `core-image-minimal` image plus a real-time test suite and tools appropriate for real-time use.
- `core-image-rt-sdk`: A `core-image-rt` image that includes everything in the cross-toolchain. The image also includes development headers and libraries to form a complete stand-alone SDK and is suitable for development using the target.
- `core-image-sato`: An image with Sato support, a mobile environment and visual style that works well with mobile devices. The image supports X11 with a Sato theme and applications such as a terminal, editor, file manager, media player, and so forth.
- `core-image-sato-dev`: A `core-image-sato` image suitable for development using the host. The image includes libraries needed to build applications on the device itself, testing and profiling tools, and debug symbols. This image was formerly `core-image-sdk`.
- `core-image-sato-sdk`: A `core-image-sato` image that includes everything in the cross-toolchain. The image also includes development headers and libraries to form a complete standalone SDK and is suitable for development using the target.
- `core-image-testmaster`: A "master" image designed to be used for automated runtime testing. Provides a "known good" image that is deployed to a separate partition so that you can boot into it and use it to deploy a second image to be tested. You can find more information about runtime testing in the "[Performing Automated Runtime Testing](#)" section in the Yocto Project Development Tasks Manual.
- `core-image-testmaster-initramfs`: A RAM-based Initial Root Filesystem (initramfs) image tailored for use with the `core-image-testmaster` image.
- `core-image-weston`: A very basic Wayland image with a terminal. This image provides the Wayland protocol libraries and the reference Weston compositor. For more information, see the "[Using Wayland and Weston](#)" section in the Yocto Project Development Tasks Manual.
- `core-image-x11`: A very basic X11 image with a terminal.

## Chapter 32. Features¶

### [32.1. Machine Features](#)

### [32.2. Distro Features](#)

### [32.3. Image Features](#)

### [32.4. Feature Backfilling](#)

This chapter provides a reference of shipped machine and distro features you can include as part of your image, a reference on image features you can select, and a reference on feature backfilling.

Features provide a mechanism for working out which packages should be included in the generated images. Distributions can select which features they want to support through the `DISTRO_FEATURES` variable, which is set or appended to in a distribution's configuration file such as `poky.conf`, `poky-tiny.conf`, `poky-lsb.conf` and so forth. Machine features are set in the `MACHINE_FEATURES` variable, which is set in the machine configuration file and specifies the hardware features for a given machine.

These two variables combine to work out which kernel modules, utilities, and other packages to include. A given distribution can support a selected subset of features so some machine features might not be included if the distribution itself does not support them.

One method you can use to determine which recipes are checking to see if a particular feature is contained or not is to `grep` through the `Metadata` for the feature. Here is an example that discovers the recipes whose build is potentially changed based on a given feature:

```
$ cd poky
$ git grep 'contains.*MACHINE_FEATURES.*feature'
```

## 32.1. Machine Features¶

The items below are features you can use with `MACHINE_FEATURES`. Features do not have a one-to-one correspondence to packages, and they can go beyond simply controlling the installation of a package or packages. Sometimes a feature can influence how certain recipes are built. For example, a feature might determine whether a particular configure option is specified within the `do_configure` task for a particular recipe.

This feature list only represents features as shipped with the Yocto Project metadata:

- **acpi**: Hardware has ACPI (x86/x86\_64 only)
- **alsa**: Hardware has ALSA audio drivers
- **apm**: Hardware uses APM (or APM emulation)
- **bluetooth**: Hardware has integrated BT
- **efi**: Support for booting through EFI
- **ext2**: Hardware HDD or Microdrive
- **keyboard**: Hardware has a keyboard
- **pcbios**: Support for booting through BIOS
- **pci**: Hardware has a PCI bus
- **pcmcia**: Hardware has PCMCIA or CompactFlash sockets

- **phone:** Mobile phone (voice) support
- **qvga:** Machine has a QVGA (320x240) display
- **rtc:** Machine has a Real-Time Clock
- **screen:** Hardware has a screen
- **serial:** Hardware has serial support (usually RS232)
- **touchscreen:** Hardware has a touchscreen
- **usbgadget:** Hardware is USB gadget device capable
- **usbhost:** Hardware is USB Host capable
- **vfat:** FAT file system support
- **wifi:** Hardware has integrated WiFi

## 32.2. Distro Features¶

The items below are features you can use with `DISTRO_FEATURES` to enable features across your distribution. Features do not have a one-to-one correspondence to packages, and they can go beyond simply controlling the installation of a package or packages. In most cases, the presence or absence of a feature translates to the appropriate option supplied to the configure script during the `do_configure` task for the recipes that optionally support the feature.

Some distro features are also machine features. These select features make sense to be controlled both at the machine and distribution configuration level. See the `COMBINED_FEATURES` variable for more information.

This list only represents features as shipped with the Yocto Project metadata:

- **alsa:** Include ALSA support (OSS compatibility kernel modules installed if available).
- **api-documentation:** Enables generation of API documentation during recipe builds. The resulting documentation is added to SDK tarballs when the `bitbake -c populate_sdk` command is used. See the "[Adding API Documentation to the Standard SDK](#)" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.
- **bluetooth:** Include bluetooth support (integrated BT only).
- **cramfs:** Include CramFS support.
- **directfb:** Include DirectFB support.
- **ext2:** Include tools for supporting for devices with internal HDD/Microdrive for storing files (instead of Flash only devices).
- **ipsec:** Include IPsec support.
- **ipv6:** Include IPv6 support.
- **keyboard:** Include keyboard support (e.g. keymaps will be loaded during boot).
- **ldconfig:** Include support for `ldconfig` and `ld.so.conf` on the target.
- **nfs:** Include NFS client support (for mounting NFS exports on device).
- **opengl:** Include the Open Graphics Library, which is a cross-language, multi-platform application programming interface used for rendering two and three-dimensional graphics.
- **pci:** Include PCI bus support.
- **pcmcia:** Include PCMCIA/CompactFlash support.
- **ppp:** Include PPP dialup support.
- **pptest:** Enables building the package tests where supported by individual recipes. For more information on package tests, see the "[Testing Packages With ptest](#)" section in the Yocto Project Development Tasks Manual.
- **smbfs:** Include SMB networks client support (for mounting Samba/Microsoft Windows shares on device).
- **systemd:** Include support for this `init` manager, which is a full replacement of for `init` with parallel starting of services, reduced shell overhead, and other features. This `init` manager is used by many distributions.
- **usbgadget:** Include USB Gadget Device support (for USB networking/serial/storage).
- **usbhost:** Include USB Host support (allows to connect external keyboard, mouse, storage, network etc).
- **wayland:** Include the Wayland display server protocol and the library that supports it.
- **wifi:** Include WiFi support (integrated only).
- **x11:** Include the X server and libraries.

## 32.3. Image Features¶

The contents of images generated by the OpenEmbedded build system can be controlled by the `IMAGE_FEATURES` and `EXTRA_IMAGE_FEATURES` variables that you typically configure in your image recipes. Through these variables, you can add several different predefined packages such as development utilities or packages with debug information needed to investigate application problems or profile applications.

The following image features are available for all images:

- **allow-empty-password:** Allows Dropbear and OpenSSH to accept root logins and logins from accounts having an empty password string.
- **dbg-pkgs:** Installs debug symbol packages for all packages installed in a given image.
- **debug-tweaks:** Makes an image suitable for development (e.g. allows root logins without passwords and enables post-installation logging). See the 'allow-empty-password', 'empty-root-password', and 'post-install-logging' features in this list for additional information.
- **dev-pkgs:** Installs development packages (headers and extra library links) for all packages installed in a given image.
- **doc-pkgs:** Installs documentation packages for all packages installed in a given image.
- **empty-root-password:** Sets the root password to an empty string, which allows logins with a blank password.
- **package-management:** Installs package management tools and preserves the package manager database.
- **post-install-logging:** Enables logging postinstall script runs to the `/var/log/postinstall.log` file on first boot of the image on the target system.

## Note

To make the `/var/log` directory on the target persistent, use the `VOLATILE_LOG_DIR` variable by setting it to "no".

- **pptest-pkgs:** Installs ptest packages for all ptest-enabled recipes.
- **read-only-rootfs:** Creates an image whose root filesystem is read-only. See the "[Creating a Read-Only Root Filesystem](#)" section in the Yocto Project Development Tasks Manual for more information.
- **splash:** Enables showing a splash screen during boot. By default, this screen is provided by `psplash`, which does allow customization. If you prefer to use an alternative splash screen package, you can do so by setting the `SPLASH` variable to a different package name (or names) within the image recipe or at the distro configuration level.
- **staticdev-pkgs:** Installs static development packages, which are static libraries (i.e. \*.a files), for all packages installed in a given image.

Some image features are available only when you inherit the `core-image` class. The current list of these valid features is as follows:

- **hwcodecs:** Installs hardware acceleration codecs.
- **nfs-server:** Installs an NFS server.
- **perf:** Installs profiling tools such as `perf`, `systemtap`, and `LTtng`. For general information on user-space tools, see the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.
- **ssh-server-dropbear:** Installs the Dropbear minimal SSH server.
- **ssh-server-openssh:** Installs the OpenSSH SSH server, which is more full-featured than Dropbear. Note that if both the OpenSSH SSH server and the Dropbear minimal SSH server are present in `IMAGE_FEATURES`, then OpenSSH will take precedence and Dropbear will not be installed.
- **tools-debug:** Installs debugging tools such as `strace` and `gdb`. For information on GDB, see the "[Debugging With the GNU Project Debugger \(GDB\) Remotely](#)," section in the Yocto Project Development Tasks Manual. For information on tracing and profiling, see the Yocto Project Profiling and Tracing Manual.
- **tools-sdk:** Installs a full SDK that runs on the device.
- **tools-testapps:** Installs device testing tools (e.g. touchscreen debugging).
- **x11:** Installs the X server.
- **x11-base:** Installs the X server with a minimal environment.
- **x11-sato:** Installs the OpenedHand Sato environment.

## 32.4. Feature Backfilling¶

Sometimes it is necessary in the OpenEmbedded build system to extend `MACHINE_FEATURES` or `DISTRO_FEATURES` to control functionality that was previously enabled and not able to be disabled. For these cases, we need to add an additional feature item to appear in one of these variables, but we do not want to force developers who have existing values of the variables in their configuration to add the new feature in order to retain the same overall level of functionality. Thus, the OpenEmbedded build system has a mechanism to automatically "backfill" these added features into existing distro or machine configurations. You can see the list of features for which this is done by finding the `DISTRO_FEATURES_BACKFILL` and `MACHINE_FEATURES_BACKFILL` variables in the `meta/conf/bitbake.conf` file.

Because such features are backfilled by default into all configurations as described in the previous paragraph, developers who wish to disable the new features need to be able to selectively prevent the backfilling from occurring. They can do this by adding the undesired feature or features to the `DISTRO_FEATURES_BACKFILL_CONSIDERED` or `MACHINE_FEATURES_BACKFILL_CONSIDERED` variables for distro features and machine features respectively.

Here are two examples to help illustrate feature backfilling:

- **The "pulseaudio" distro feature option:** Previously, PulseAudio support was enabled within the Qt and GStreamer frameworks. Because of this, the feature is backfilled and thus enabled for all distros through the `DISTRO_FEATURES_BACKFILL` variable in the `meta/conf/bitbake.conf` file. However, your distro needs to disable the feature. You can disable the feature without affecting other existing distro configurations that need PulseAudio support by adding "pulseaudio" to

`DISTRO_FEATURES_BACKFILL_CONSIDERED` in your distro's `.conf` file. Adding the feature to this variable when it also exists in the `DISTRO_FEATURES_BACKFILL` variable prevents the build system from adding the feature to your configuration's `DISTRO_FEATURES`, effectively disabling the feature for that particular distro.

- **The "rtc" machine feature option:** Previously, real time clock (RTC) support was enabled for all target devices. Because of this, the feature is backfilled and thus enabled for all machines through the `MACHINE_FEATURES_BACKFILL` variable in the `meta/conf/bitbake.conf` file. However, your target device does not have this capability. You can disable RTC support for your device without affecting other machines that need RTC support by adding the feature to your machine's `MACHINE_FEATURES_BACKFILL_CONSIDERED` list in the machine's `.conf` file. Adding the feature to this variable when it also exists in the `MACHINE_FEATURES_BACKFILL` variable prevents the build system from adding the feature to your configuration's `MACHINE_FEATURES`, effectively disabling RTC support for that particular machine.

## Chapter 33. Variables Glossary¶

### Glossary

This chapter lists common variables used in the OpenEmbedded build system and gives an overview of their function and contents.

### Glossary¶

A B C D E F G H I K L M N O P R S T U V W X

#### A

##### ABIEXTENSION¶

Extension to the Application Binary Interface (ABI) field of the GNU canonical architecture name (e.g. "eabi").

ABI extensions are set in the machine include files. For example, the `meta/conf/machine/include/arm/arch-arm.inc` file sets the following extension:

```
ABIEXTENSION = "eabi"
```

##### ALLOW\_EMPTY¶

Specifies whether to produce an output package even if it is empty. By default, BitBake does not produce empty packages. This default behavior can cause issues when there is an [RDEPENDS](#) or some other hard runtime requirement on the existence of the package.

Like all package-controlling variables, you must always use them in conjunction with a package name override, as in:

```
ALLOW_EMPTY_${PN} = "1"
ALLOW_EMPTY_${PN}-dev = "1"
ALLOW_EMPTY_${PN}-staticdev = "1"
```

##### ALTERNATIVE¶

Lists commands in a package that need an alternative binary naming scheme. Sometimes the same command is provided in multiple packages. When this occurs, the OpenEmbedded build system needs to use the alternatives system to create a different binary naming scheme so the commands can co-exist.

To use the variable, list out the package's commands that also exist as part of another package. For example, if the `busybox` package has four commands that also exist as part of another package, you identify them as follows:

```
ALTERNATIVE_busybox = "sh sed test bracket"
```

For more information on the alternatives system, see the ["update-alternatives.bbclass"](#) section.

##### ALTERNATIVE\_LINK\_NAME¶

Used by the alternatives system to map duplicated commands to actual locations. For example, if the `bracket` command provided by the `busybox` package is duplicated through another package, you must use the `ALTERNATIVE_LINK_NAME` variable to specify the actual location:

```
ALTERNATIVE_LINK_NAME[bracket] = "/usr/bin/["
```

In this example, the binary for the `bracket` command (i.e. `[`) from the `busybox` package resides in `/usr/bin/`.

### Note

If `ALTERNATIVE_LINK_NAME` is not defined, it defaults to `${bindir}/name`.

For more information on the alternatives system, see the ["update-alternatives.bbclass"](#) section.

##### ALTERNATIVE\_PRIORITY¶

Used by the alternatives system to create default priorities for duplicated commands. You can use the variable to create a single default regardless of the command name or package, a default for specific duplicated commands regardless of the package, or a default for specific commands tied to particular packages. Here are the available syntax forms:

```
ALTERNATIVE_PRIORITY = "priority"
ALTERNATIVE_PRIORITY[name] = "priority"
ALTERNATIVE_PRIORITY_pkg[name] = "priority"
```

For more information on the alternatives system, see the ["update-alternatives.bbclass"](#) section.

ALTERNATIVE\_TARGET¶

Used by the alternatives system to create default link locations for duplicated commands. You can use the variable to create a single default location for all duplicated commands regardless of the command name or package, a default for specific duplicated commands regardless of the package, or a default for specific commands tied to particular packages. Here are the available syntax forms:

```
ALTERNATIVE_TARGET = "target"
ALTERNATIVE_TARGET[name] = "target"
ALTERNATIVE_TARGET_pkg[name] = "target"
```

Note

If `ALTERNATIVE_TARGET` is not defined, it inherits the value from the `ALTERNATIVE_LINK_NAME` variable.

If `ALTERNATIVE_LINK_NAME` and `ALTERNATIVE_TARGET` are the same, the target for `ALTERNATIVE_TARGET` has `".{BPN}"` appended to it.

Finally, if the file referenced has not been renamed, the alternatives system will rename it to avoid the need to rename alternative files in the `do_install` task while retaining support for the command if necessary.

For more information on the alternatives system, see the ["update-alternatives.bbclass"](#) section.

APPEND¶

An override list of append strings for each target specified with `LABELS`.

See the `grub-efi` class for more information on how this variable is used.

AR¶

The minimal command and arguments used to run `ar`.

ARCHIVER\_MODE¶

When used with the `archiver` class, determines the type of information used to create a released archive. You can use this variable to create archives of patched source, original source, configured source, and so forth by employing the following variable flags (varflags):

|                                                             |                                                               |
|-------------------------------------------------------------|---------------------------------------------------------------|
| <code>ARCHIVER_MODE[src] = "original"</code>                | <code># Uses original (unpacked) # files.</code>              |
| <code>ARCHIVER_MODE[src] = "patched"</code>                 | <code># Uses patched source files # the default.</code>       |
| <code>ARCHIVER_MODE[src] = "configured"</code>              | <code># Uses configured source fi</code>                      |
| <code>ARCHIVER_MODE[diff] = "1"</code>                      | <code># Uses patches between do_u # do_patch.</code>          |
| <code>ARCHIVER_MODE[diff-exclude] ?= "file file ..."</code> | <code># Lists files and directori # exclude from diff.</code> |
| <code>ARCHIVER_MODE[dumpdata] = "1"</code>                  | <code># Uses environment data.</code>                         |
| <code>ARCHIVER_MODE[recipe] = "1"</code>                    | <code># Uses recipe and include f</code>                      |
| <code>ARCHIVER_MODE[srpm] = "1"</code>                      | <code># Uses RPM package files.</code>                        |

For information on how the variable works, see the `meta/classes/archiver.bbclass` file in the [Source Directory](#).

AS¶

Minimal command and arguments needed to run the assembler.

ASSUME\_PROVIDED¶

Lists recipe names (`P_N` values) BitBake does not attempt to build. Instead, BitBake assumes these recipes have already been built.

In OpenEmbedded-Core, `ASSUME_PROVIDED` mostly specifies native tools that should not be built. An example is `git-native`, which when specified, allows for the Git binary from the host to be used rather than building `git-native`.

ASSUME\_SHLIBS¶



Provides additional `shlibs` provider mapping information, which adds to or overwrites the information provided automatically by the system. Separate multiple entries using spaces.

As an example, use the following form to add an `shlib` provider of `shlibname` in `packagename` with the optional `version`:

```
shlibname:packagename[_version]
```

Here is an example that adds a shared library named `libEGL.so.1` as being provided by the `libegl-implementation` package:

```
ASSUME_SHLIBS = "libEGL.so.1:libegl-implementation"
```

#### AUTHOR¶

The email address used to contact the original author or authors in order to send patches and forward bugs.

#### AUTO\_LIBNAME\_PKGS¶

When the `debian` class is inherited, which is the default behavior, `AUTO_LIBNAME_PKGS` specifies which packages should be checked for libraries and renamed according to Debian library package naming.

The default value is `"${PACKAGES}"`, which causes the `debian` class to act on all packages that are explicitly generated by the recipe.

#### AUTO\_SYSLINUXMENU¶

Enables creating an automatic menu for the `syslinux` bootloader. You must set this variable in your recipe. The `syslinux` class checks this variable.

#### AUTOREV¶

When `SRCREV` is set to the value of this variable, it specifies to use the latest source revision in the repository. Here is an example:

```
SRCREV = "${AUTOREV}"
```

If you use the previous statement to retrieve the latest version of software, you need to be sure `PV` contains `${SRCPV}`. For example, suppose you have a kernel recipe that inherits the `kernel` class and you use the previous statement. In this example, `${SRCPV}` does not automatically get into `PV`. Consequently, you need to change `PV` in your recipe so that it does contain `${SRCPV}`.

For more information see the "[Automatically Incrementing a Binary Package Revision Number](#)" section in the Yocto Project Development Tasks Manual.

#### AVAILTUNES¶

The list of defined CPU and Application Binary Interface (ABI) tunings (i.e. "tunes") available for use by the OpenEmbedded build system.

The list simply presents the tunes that are available. Not all tunes may be compatible with a particular machine configuration, or with each other in a `Multilib` configuration.

To add a tune to the list, be sure to append it with spaces using the `"+="` BitBake operator. Do not simply replace the list by using the `"="` operator. See the "[Basic Syntax](#)" section in the BitBake User Manual for more information.

## B

#### B¶

The directory within the `Build Directory` in which the OpenEmbedded build system places generated objects during a recipe's build process. By default, this directory is the same as the `S` directory, which is defined as:

```
S = "${WORKDIR}/${BP}"
```

You can separate the (`S`) directory and the directory pointed to by the `B` variable. Most Autotools-based recipes support separating these directories. The build system defaults to using separate directories for `gcc` and some kernel recipes.

#### BAD\_RECOMMENDATIONS¶

Lists "recommended-only" packages to not install. Recommended-only packages are packages installed only through the `RRECOMMENDS` variable. You can prevent any of these "recommended" packages from being installed by listing them with the `BAD_RECOMMENDATIONS` variable:

```
BAD_RECOMMENDATIONS = "package_name package_name package_name ..."
```

You can set this variable globally in your `local.conf` file or you can attach it to a specific image recipe by using the recipe name override:

```
BAD_RECOMMENDATIONS_pn-target_image = "package_name"
```

It is important to realize that if you choose to not install packages using this variable and some other packages are dependent on them (i.e. listed in a recipe's `RDEPENDS` variable), the OpenEmbedded build system ignores your request and will install the packages to avoid dependency errors.

Support for this variable exists only when using the IPK and RPM packaging backend. Support does not exist for DEB.

See the `NO_RECOMMENDATIONS` and the `PACKAGE_EXCLUDE` variables for related information.

#### BASE\_LIB¶



The library directory name for the CPU or Application Binary Interface (ABI) tune. The `BASE_LIB` applies only in the Multilib context. See the "[Combining Multiple Versions of Library Files into One Image](#)" section in the Yocto Project Development Tasks Manual for information on Multilib.

The `BASE_LIB` variable is defined in the machine include files in the [Source Directory](#). If Multilib is not being used, the value defaults to "lib".

`BASE_WORKDIR`

Points to the base of the work directory for all recipes. The default value is `"${TMPDIR}/work"`.

`BB_ALLOWED_NETWORKS`

Specifies a space-delimited list of hosts that the fetcher is allowed to use to obtain the required source code. Following are considerations surrounding this variable:

- This host list is only used if `BB_NO_NETWORK` is either not set or set to "0".
- Limited support for wildcard matching against the beginning of host names exists. For example, the following setting matches `git.gnu.org`, `ftp.gnu.org`, and `foo.git.gnu.org`.

```
BB_ALLOWED_NETWORKS = "*.gnu.org"
```

## Important

The use of the "\*" character only works at the beginning of a host name and it must be isolated from the remainder of the host name. You cannot use the wildcard character in any other location of the name or combined with the front part of the name.

For example, `*.foo.bar` is supported, while `*aa.foo.bar` is not.

- Mirrors not in the host list are skipped and logged in debug.
- Attempts to access networks not in the host list cause a failure.

Using `BB_ALLOWED_NETWORKS` in conjunction with `PREMIRRORS` is very useful. Adding the host you want to use to `PREMIRRORS` results in the source code being fetched from an allowed location and avoids raising an error when a host that is not allowed is in a `SRC_URI` statement. This is because the fetcher does not attempt to use the host listed in `SRC_URI` after a successful fetch from the `PREMIRRORS` occurs.

`BB_DANGLINGAPPENDS_WARNONLY`

Defines how BitBake handles situations where an append file (`.bbappend`) has no corresponding recipe file (`.bb`). This condition often occurs when layers get out of sync (e.g. `oe-core` bumps a recipe version and the old recipe no longer exists and the other layer has not been updated to the new version of the recipe yet).

The default fatal behavior is safest because it is the sane reaction given something is out of sync. It is important to realize when your changes are no longer being applied.

You can change the default behavior by setting this variable to "1", "yes", or "true" in your `local.conf` file, which is located in the [Build Directory](#). Here is an example:

```
BB_DANGLINGAPPENDS_WARNONLY = "1"
```

`BB_DISKMON_DIRS`

Monitors disk space and available inodes during the build and allows you to control the build based on these parameters.

Disk space monitoring is disabled by default. To enable monitoring, add the `BB_DISKMON_DIRS` variable to your `conf/local.conf` file found in the [Build Directory](#). Use the following form:

```
BB_DISKMON_DIRS = "action,dir,threshold [...]"
```

where:

*action* is:

```
ABORT: Immediately abort the build when
 a threshold is broken.
STOPTASKS: Stop the build after the currently
 executing tasks have finished when
 a threshold is broken.
WARN: Issue a warning but continue the
 build when a threshold is broken.
 Subsequent warnings are issued as
 defined by the BB_DISKMON_WARNINTERVAL
 variable, which must be defined in
 the conf/local.conf file.
```

*dir* is:

```
Any directory you choose. You can specify one or
more directories to monitor by separating the
groupings with a space. If two directories are
on the same device, only the first directory
is monitored.
```

*threshold* is:  
 Either the minimum available disk space,  
 the minimum number of free inodes, or  
 both. You must specify at least one. To  
 omit one or the other, simply omit the value.  
 Specify the threshold using G, M, K for Gbytes,  
 Mbytes, and Kbytes, respectively. If you do  
 not specify G, M, or K, Kbytes is assumed by  
 default. Do not use GB, MB, or KB.

Here are some examples:

```
BB_DISKMON_DIRS = "ABORT,${TMPDIR},1G,100K WARN,${SSTATE_DIR},1G,100K"
BB_DISKMON_DIRS = "STOPTASKS,${TMPDIR},1G"
BB_DISKMON_DIRS = "ABORT,${TMPDIR},,100K"
```

The first example works only if you also provide the `BB_DISKMON_WARNINTERVAL` variable in the `conf/local.conf`. This example causes the build system to immediately abort when either the disk space in `${TMPDIR}` drops below 1 Gbyte or the available free inodes drops below 100 Kbytes. Because two directories are provided with the variable, the build system also issue a warning when the disk space in the `${SSTATE_DIR}` directory drops below 1 Gbyte or the number of free inodes drops below 100 Kbytes. Subsequent warnings are issued during intervals as defined by the `BB_DISKMON_WARNINTERVAL` variable.

The second example stops the build after all currently executing tasks complete when the minimum disk space in the `${TMPDIR}` directory drops below 1 Gbyte. No disk monitoring occurs for the free inodes in this case.

The final example immediately aborts the build when the number of free inodes in the `${TMPDIR}` directory drops below 100 Kbytes. No disk space monitoring for the directory itself occurs in this case.

#### `BB_DISKMON_WARNINTERVAL`

Defines the disk space and free inode warning intervals. To set these intervals, define the variable in your `conf/local.conf` file in the [Build Directory](#).

If you are going to use the `BB_DISKMON_WARNINTERVAL` variable, you must also use the `BB_DISKMON_DIRS` variable and define its action as "WARN". During the build, subsequent warnings are issued each time disk space or number of free inodes further reduces by the respective interval.

If you do not provide a `BB_DISKMON_WARNINTERVAL` variable and you do use `BB_DISKMON_DIRS` with the "WARN" action, the disk monitoring interval defaults to the following:

```
BB_DISKMON_WARNINTERVAL = "50M,5K"
```

When specifying the variable in your configuration file, use the following form:

```
BB_DISKMON_WARNINTERVAL = "disk_space_interval,disk_inode_interval"
```

where:

*disk\_space\_interval* is:  
 An interval of memory expressed in either  
 G, M, or K for Gbytes, Mbytes, or Kbytes,  
 respectively. You cannot use GB, MB, or KB.

*disk\_inode\_interval* is:  
 An interval of free inodes expressed in either  
 G, M, or K for Gbytes, Mbytes, or Kbytes,  
 respectively. You cannot use GB, MB, or KB.

Here is an example:

```
BB_DISKMON_DIRS = "WARN,${SSTATE_DIR},1G,100K"
BB_DISKMON_WARNINTERVAL = "50M,5K"
```

These variables cause the OpenEmbedded build system to issue subsequent warnings each time the available disk space further reduces by 50 Mbytes or the number of free inodes further reduces by 5 Kbytes in the `${SSTATE_DIR}` directory. Subsequent warnings based on the interval occur each time a respective interval is reached beyond the initial warning (i.e. 1 Gbytes and 100 Kbytes).

#### `BB_GENERATE_MIRROR_TARBALLS`

Causes tarballs of the source control repositories (e.g. Git repositories), including metadata, to be placed in the `DL_DIR` directory.

For performance reasons, creating and placing tarballs of these repositories is not the default action by the OpenEmbedded build system.

```
BB_GENERATE_MIRROR_TARBALLS = "1"
```

Set this variable in your `local.conf` file in the [Build Directory](#).

Once you have the tarballs containing your source files, you can clean up your `DL_DIR` directory by deleting any Git or other source control work directories.

#### `BB_NUMBER_THREADS`

The maximum number of tasks BitBake should run in parallel at any one time. The OpenEmbedded build system automatically configures this variable to be equal to the number of cores on the build system. For example, a system with a dual core

processor that also uses hyper-threading causes the `BB_NUMBER_THREADS` variable to default to "4".

For single socket systems (i.e. one CPU), you should not have to override this variable to gain optimal parallelism during builds. However, if you have very large systems that employ multiple physical CPUs, you might want to make sure the `BB_NUMBER_THREADS` variable is not set higher than "20".

For more information on speeding up builds, see the "[Speeding Up a Build](#)" section in the Yocto Project Development Tasks Manual.

#### `BB_SERVER_TIMEOUT`

Specifies the time (in seconds) after which to unload the BitBake server due to inactivity. Set `BB_SERVER_TIMEOUT` to determine how long the BitBake server stays resident between invocations.

For example, the following statement in your `local.conf` file instructs the server to be unloaded after 20 seconds of inactivity:

```
BB_SERVER_TIMEOUT = "20"
```

If you want the server to never be unloaded, set `BB_SERVER_TIMEOUT` to "-1".

#### `BBCLASSEXTEND`

Allows you to extend a recipe so that it builds variants of the software. Common variants for recipes exist such as "natives" like `quilt-native`, which is a copy of Quilt built to run on the build system; "crosses" such as `gcc-cross`, which is a compiler built to run on the build machine but produces binaries that run on the target [MACHINE](#); "nativesdk", which targets the SDK machine instead of `MACHINE`; and "multilibs" in the form `multilib:multilib_name`.

To build a different variant of the recipe with a minimal amount of code, it usually is as simple as adding the following to your recipe:

```
BBCLASSEXTEND += "native nativesdk"
BBCLASSEXTEND += "multilib:multilib_name"
```

### Note

Internally, the `BBCLASSEXTEND` mechanism generates recipe variants by rewriting variable values and applying overrides such as `_class-native`. For example, to generate a native version of a recipe, a `DEPENDS` on "foo" is rewritten to a `DEPENDS` on "foo-native".

Even when using `BBCLASSEXTEND`, the recipe is only parsed once. Parsing once adds some limitations. For example, it is not possible to include a different file depending on the variant, since `include` statements are processed when the recipe is parsed.

#### `BBFILE_COLLECTIONS`

Lists the names of configured layers. These names are used to find the other `BBFILE_*` variables. Typically, each layer will append its name to this variable in its `conf/layer.conf` file.

#### `BBFILE_PATTERN`

Variable that expands to match files from `BBFILES` in a particular layer. This variable is used in the `conf/layer.conf` file and must be suffixed with the name of the specific layer (e.g. `BBFILE_PATTERN_emenlow`).

#### `BBFILE_PRIORITY`

Assigns the priority for recipe files in each layer.

This variable is useful in situations where the same recipe appears in more than one layer. Setting this variable allows you to prioritize a layer against other layers that contain the same recipe - effectively letting you control the precedence for the multiple layers. The precedence established through this variable stands regardless of a recipe's version (`PV` variable). For example, a layer that has a recipe with a higher `PV` value but for which the `BBFILE_PRIORITY` is set to have a lower precedence still has a lower precedence.

A larger value for the `BBFILE_PRIORITY` variable results in a higher precedence. For example, the value 6 has a higher precedence than the value 5. If not specified, the `BBFILE_PRIORITY` variable is set based on layer dependencies (see the `LAYERDEPENDS` variable for more information. The default priority, if unspecified for a layer with no dependencies, is the lowest defined priority + 1 (or 1 if no priorities are defined).

### Tip

You can use the command `bitbake-layers show-layers` to list all configured layers along with their priorities.

|                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>BBFILES</b>             | <p>A space-separated list of recipe files BitBake uses to build software.</p> <p>When specifying recipe files, you can pattern match using Python's <code>glob</code> syntax. For details on the syntax, see the documentation by following the previous link.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>BBFILES_DYNAMIC</b>     | <p>Activates content when identified layers are present. You identify the layers by the collections that the layers define.</p> <p>Use the <code>BBFILES_DYNAMIC</code> variable to avoid <code>.bbappend</code> files whose corresponding <code>.bb</code> file is in a layer that attempts to modify other layers through <code>.bbappend</code> but does not want to introduce a hard dependency on those other layers.</p> <p>Use the following form for <code>BBFILES_DYNAMIC</code>:</p> <pre>collection_name:filename_pattern</pre> <p>The following example identifies two collection names and two filename patterns:</p> <pre>BBFILES_DYNAMIC += " \ clang-layer:\${LAYERDIR}/bbappends/meta-clang/*/*/*.bbappend \ core:\${LAYERDIR}/bbappends/openembedded-core/meta/*/*/*.bbappend \ "</pre> <p>This next example shows an error message that occurs because invalid entries are found, which cause parsing to abort:</p> <pre>ERROR: BBFILES_DYNAMIC entries must be of the form &lt;collection name&gt;:&lt;filename&gt; /work/my-layer/bbappends/meta-security-isafw/*/*/*.bbappend /work/my-layer/bbappends/openembedded-core/meta/*/*/*.bbappend</pre>                                                                                                                                                                                                                                                                              |
| <b>BBINCLUDELOGS</b>       | <p>Variable that controls how BitBake displays logs on build failure.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>BBINCLUDELOGS_LINES</b> | <p>If <code>BBINCLUDELOGS</code> is set, specifies the maximum number of lines from the task log file to print when reporting a failed task. If you do not set <code>BBINCLUDELOGS_LINES</code>, the entire log is printed.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>BBLAYERS</b>            | <p>Lists the layers to enable during the build. This variable is defined in the <code>bblayers.conf</code> configuration file in the <a href="#">Build Directory</a>. Here is an example:</p> <pre>BBLAYERS = " \ /home/scottrif/poky/meta \ /home/scottrif/poky/meta-poky \ /home/scottrif/poky/meta-yocto-bsp \ /home/scottrif/poky/meta-mykernel \ "</pre> <p>This example enables four layers, one of which is a custom, user-defined layer named <code>meta-mykernel</code>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>BBMASK</b>              | <p>Prevents BitBake from processing recipes and recipe append files.</p> <p>You can use the <code>BBMASK</code> variable to "hide" these <code>.bb</code> and <code>.bbappend</code> files. BitBake ignores any recipe or recipe append files that match any of the expressions. It is as if BitBake does not see them at all. Consequently, matching files are not parsed or otherwise used by BitBake.</p> <p>The values you provide are passed to Python's regular expression compiler. Consequently, the syntax follows Python's Regular Expression (re) syntax. The expressions are compared against the full paths to the files. For complete syntax information, see Python's documentation at <a href="http://docs.python.org/3/library/re.html#re">http://docs.python.org/3/library/re.html#re</a>.</p> <p>The following example uses a complete regular expression to tell BitBake to ignore all recipe and recipe append files in the <code>meta-ti/recipes-misc/</code> directory:</p> <pre>BBMASK = "meta-ti/recipes-misc/"</pre> <p>If you want to mask out multiple directories or recipes, you can specify multiple regular expression fragments. This next example masks out multiple directories and individual recipes:</p> <pre>BBMASK += "/meta-ti/recipes-misc/ meta-ti/recipes-ti/packagegroup/" BBMASK += "/meta-oe/recipes-support/" BBMASK += "/meta-foo/*.*/openldap" BBMASK += "opencv.*\bbappend" BBMASK += "lzma"</pre> |

## Note

When specifying a directory name, use the trailing slash character to ensure you match just that directory name.

|                      |                                                                                                                                                                            |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>BBMULTICONFIG</b> | <p>Specifies each additional separate configuration when you are building targets with multiple configurations. Use this variable in your <code>conf/local.conf</code></p> |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

configuration file. Specify a *multiconfigname* for each configuration file you are using. For example, the following line specifies three configuration files:

```
BBMULTICONFIG = "configA configB configC"
```

Each configuration file you use must reside in the [Build Directory](#) `conf/multiconfig` directory (e.g. `build_directory/conf/multiconfig/configA.conf`).

For information on how to use BBMULTICONFIG in an environment that supports building targets with multiple configurations, see the "[Building Images for Multiple Targets Using Multiple Configurations](#)" section in the Yocto Project Development Tasks Manual.

#### BBPATH¶

Used by BitBake to locate `.bbclass` and configuration files. This variable is analogous to the `PATH` variable.

### Note

If you run BitBake from a directory outside of the [Build Directory](#), you must be sure to set BBPATH to point to the Build Directory. Set the variable as you would any environment variable and then run BitBake:

```
$ BBPATH = "build_directory"
$ export BBPATH
$ bitbake target
```

#### BBSERVER¶

If defined in the BitBake environment, BBSERVER points to the BitBake remote server.

Use the following format to export the variable to the BitBake environment:

```
export BBSERVER=localhost:$port"
```

By default, BBSERVER also appears in [BB\\_HASHBASE\\_WHITELIST](#). Consequently, BBSERVER is excluded from checksum and dependency data.

#### BINCONFIG¶

When inheriting the [binconfig-disabled](#) class, this variable specifies binary configuration scripts to disable in favor of using `pkg-config` to query the information. The `binconfig-disabled` class will modify the specified scripts to return an error so that calls to them can be easily found and replaced.

To add multiple scripts, separate them by spaces. Here is an example from the `libpng` recipe:

```
BINCONFIG = "${bindir}/libpng-config ${bindir}/libpng16-config"
```

#### BINCONFIG\_GLOB¶

When inheriting the [binconfig](#) class, this variable specifies a wildcard for configuration scripts that need editing. The scripts are edited to correct any paths that have been set up during compilation so that they are correct for use when installed into the `sysroot` and called by the build processes of other recipes.

### Note

The `BINCONFIG_GLOB` variable uses [shell globbing](#), which is recognition and expansion of wildcards during pattern matching. Shell globbing is very similar to [fnmatch](#) and [glob](#).

For more information on how this variable works, see `meta/classes/binconfig.bbclass` in the [Source Directory](#). You can also find general information on the class in the "[binconfig.bbclass](#)" section.

#### BP¶

The base recipe name and version but without any special recipe name suffix (i.e. `-native`, `lib64-`, and so forth). BP is comprised of the following:

```
${BPN}-${PV}
```

#### BPN¶

This variable is a version of the [PN](#) variable with common prefixes and suffixes removed, such as `nativesdk-`, `-cross`, `-native`, and `multilib's lib64-` and `lib32-`. The exact lists of prefixes and suffixes removed are specified by the [MLPREFIX](#) and [SPECIAL\\_PKGSUFFIX](#) variables, respectively.

#### BUGTRACKER¶

Specifies a URL for an upstream bug tracking website for a recipe. The OpenEmbedded build system does not use this variable. Rather, the variable is a useful pointer in case a bug in the software being built needs to be manually reported.

#### BUILD\_ARCH¶

|                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-----------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                         | Specifies the architecture of the build host (e.g. <code>i686</code> ). The OpenEmbedded build system sets the value of <code>BUILD_ARCH</code> from the machine name reported by the <code>uname</code> command.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <code>BUILD_AS_ARCH</code>              | Specifies the architecture-specific assembler flags for the build host. By default, the value of <code>BUILD_AS_ARCH</code> is empty.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <code>BUILD_CC_ARCH</code>              | Specifies the architecture-specific C compiler flags for the build host. By default, the value of <code>BUILD_CC_ARCH</code> is empty.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <code>BUILD_CCLD</code>                 | Specifies the linker command to be used for the build host when the C compiler is being used as the linker. By default, <code>BUILD_CCLD</code> points to GCC and passes as arguments the value of <code>BUILD_CC_ARCH</code> , assuming <code>BUILD_CC_ARCH</code> is set.                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <code>BUILD_CFLAGS</code>               | Specifies the flags to pass to the C compiler when building for the build host. When building in the <code>-native</code> context, <code>CFLAGS</code> is set to the value of this variable by default.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <code>BUILD_CPPFLAGS</code>             | Specifies the flags to pass to the C preprocessor (i.e. to both the C and the C++ compilers) when building for the build host. When building in the <code>-native</code> context, <code>CPPFLAGS</code> is set to the value of this variable by default.                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <code>BUILD_CXXFLAGS</code>             | Specifies the flags to pass to the C++ compiler when building for the build host. When building in the <code>-native</code> context, <code>CXXFLAGS</code> is set to the value of this variable by default.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <code>BUILD_FC</code>                   | Specifies the Fortran compiler command for the build host. By default, <code>BUILD_FC</code> points to Gfortran and passes as arguments the value of <code>BUILD_CC_ARCH</code> , assuming <code>BUILD_CC_ARCH</code> is set.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <code>BUILD_LD</code>                   | Specifies the linker command for the build host. By default, <code>BUILD_LD</code> points to the GNU linker ( <code>ld</code> ) and passes as arguments the value of <code>BUILD_LD_ARCH</code> , assuming <code>BUILD_LD_ARCH</code> is set.                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <code>BUILD_LD_ARCH</code>              | Specifies architecture-specific linker flags for the build host. By default, the value of <code>BUILD_LD_ARCH</code> is empty.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <code>BUILD_LDFLAGS</code>              | Specifies the flags to pass to the linker when building for the build host. When building in the <code>-native</code> context, <code>LDFLAGS</code> is set to the value of this variable by default.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <code>BUILD_OPTIMIZATION</code>         | Specifies the optimization flags passed to the C compiler when building for the build host or the SDK. The flags are passed through the <code>BUILD_CFLAGS</code> and <code>BUILDSDK_CFLAGS</code> default values.<br><br>The default value of the <code>BUILD_OPTIMIZATION</code> variable is <code>"-O2 -pipe"</code> .                                                                                                                                                                                                                                                                                                                                                                                                |
| <code>BUILD_OS</code>                   | Specifies the operating system in use on the build host (e.g. <code>"linux"</code> ). The OpenEmbedded build system sets the value of <code>BUILD_OS</code> from the OS reported by the <code>uname</code> command - the first word, converted to lower-case characters.                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <code>BUILD_PREFIX</code>               | The toolchain binary prefix used for native recipes. The OpenEmbedded build system uses the <code>BUILD_PREFIX</code> value to set the <code>TARGET_PREFIX</code> when building for native recipes.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <code>BUILD_STRIP</code>                | Specifies the command to be used to strip debugging symbols from binaries produced for the build host. By default, <code>BUILD_STRIP</code> points to <code>\${BUILD_PREFIX}strip</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <code>BUILD_SYS</code>                  | Specifies the system, including the architecture and the operating system, to use when building for the build host (i.e. when building <code>native</code> recipes).<br><br>The OpenEmbedded build system automatically sets this variable based on <code>BUILD_ARCH</code> , <code>BUILD_VENDOR</code> , and <code>BUILD_OS</code> . You do not need to set the <code>BUILD_SYS</code> variable yourself.                                                                                                                                                                                                                                                                                                               |
| <code>BUILD_VENDOR</code>               | Specifies the vendor name to use when building for the build host. The default value is an empty string ( <code>""</code> ).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <code>BUILDDIR</code>                   | Points to the location of the <a href="#">Build Directory</a> . You can define this directory indirectly through the <code>oe-init-build-env</code> script by passing in a Build Directory path when you run the script. If you run the script and do not provide a Build Directory path, the <code>BUILDDIR</code> defaults to <code>build</code> in the current directory.                                                                                                                                                                                                                                                                                                                                             |
| <code>BUILDHISTORY_COMMIT</code>        | When inheriting the <a href="#">buildhistory</a> class, this variable specifies whether or not to commit the build history output in a local Git repository. If set to <code>"1"</code> , this local repository will be maintained automatically by the <code>buildhistory</code> class and a commit will be created on every build for changes to each top-level subdirectory of the build history output (images, packages, and sdk). If you want to track changes to build history over time, you should set this value to <code>"1"</code> .<br><br>By default, the <code>buildhistory</code> class does not commit the build history output in a local Git repository:<br><br><pre>BUILDHISTORY_COMMIT ?= "0"</pre> |
| <code>BUILDHISTORY_COMMIT_AUTHOR</code> | When inheriting the <a href="#">buildhistory</a> class, this variable specifies the author to use for each Git commit. In order for the                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

`BUILDHISTORY_COMMIT_AUTHOR` variable to work, the `BUILDHISTORY_COMMIT` variable must be set to "1".

Git requires that the value you provide for the `BUILDHISTORY_COMMIT_AUTHOR` variable takes the form of "name email@host". Providing an email address or host that is not valid does not produce an error.

By default, the `buildhistory` class sets the variable as follows:

```
BUILDHISTORY_COMMIT_AUTHOR ?= "buildhistory <buildhistory@${DISTRO}>"
```

#### `BUILDHISTORY_DIR`

When inheriting the `buildhistory` class, this variable specifies the directory in which build history information is kept. For more information on how the variable works, see the `buildhistory.class`.

By default, the `buildhistory` class sets the directory as follows:

```
BUILDHISTORY_DIR ?= "${TOPDIR}/buildhistory"
```

#### `BUILDHISTORY_FEATURES`

When inheriting the `buildhistory` class, this variable specifies the build history features to be enabled. For more information on how build history works, see the "Maintaining Build Output Quality" section in the Yocto Project Development Tasks Manual.

You can specify these features in the form of a space-separated list:

- **image:** Analysis of the contents of images, which includes the list of installed packages among other things.
- **package:** Analysis of the contents of individual packages.
- **sdk:** Analysis of the contents of the software development kit (SDK).
- **task:** Save output file signatures for `shared state` (sstate) tasks. This saves one file per task and lists the SHA-256 checksums for each file staged (i.e. the output of the task).

By default, the `buildhistory` class enables the following features:

```
BUILDHISTORY_FEATURES ?= "image package sdk"
```

#### `BUILDHISTORY_IMAGE_FILES`

When inheriting the `buildhistory` class, this variable specifies a list of paths to files copied from the image contents into the build history directory under an "image-files" directory in the directory for the image, so that you can track the contents of each file. The default is to copy `/etc/passwd` and `/etc/group`, which allows you to monitor for changes in user and group entries. You can modify the list to include any file. Specifying an invalid path does not produce an error. Consequently, you can include files that might not always be present.

By default, the `buildhistory` class provides paths to the following files:

```
BUILDHISTORY_IMAGE_FILES ?= "/etc/passwd /etc/group"
```

#### `BUILDHISTORY_PUSH_REPO`

When inheriting the `buildhistory` class, this variable optionally specifies a remote repository to which build history pushes Git changes. In order for `BUILDHISTORY_PUSH_REPO` to work, `BUILDHISTORY_COMMIT` must be set to "1".

The repository should correspond to a remote address that specifies a repository as understood by Git, or alternatively to a remote name that you have set up manually using `git remote` within the local repository.

By default, the `buildhistory` class sets the variable as follows:

```
BUILDHISTORY_PUSH_REPO ?= ""
```

#### `BUILDSDK_CFLAGS`

Specifies the flags to pass to the C compiler when building for the SDK. When building in the `nativesdk-` context, `CFLAGS` is set to the value of this variable by default.

#### `BUILDSDK_CPPFLAGS`

Specifies the flags to pass to the C pre-processor (i.e. to both the C and the C++ compilers) when building for the SDK. When building in the `nativesdk-` context, `CPPFLAGS` is set to the value of this variable by default.

#### `BUILDSDK_CXXFLAGS`

Specifies the flags to pass to the C++ compiler when building for the SDK. When building in the `nativesdk-` context, `CXXFLAGS` is set to the value of this variable by default.

#### `BUILDSDK_LDFLAGS`

Specifies the flags to pass to the linker when building for the SDK. When building in the `nativesdk-` context, `LDFLAGS` is set to the value of this variable by default.

#### `BUILDSTATS_BASE`

Points to the location of the directory that holds build statistics when you use and enable the `buildstats` class. The `BUILDSTATS_BASE` directory defaults to `${TMPDIR}/buildstats/`.

#### `BUSYBOX_SPLIT_SUID`



For the BusyBox recipe, specifies whether to split the output executable file into two parts: one for features that require `setuid root`, and one for the remaining features (i.e. those that do not require `setuid root`).

The `BUSYBOX_SPLIT_SUID` variable defaults to "1", which results in splitting the output executable file. Set the variable to "0" to get a single output executable file.

## C

|                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>CACHE</code>         | Specifies the directory BitBake uses to store a cache of the <a href="#">Metadata</a> so it does not need to be parsed every time BitBake is started.                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <code>CC</code>            | The minimal command and arguments used to run the C compiler.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <code>CFLAGS</code>        | Specifies the flags to pass to the C compiler. This variable is exported to an environment variable and thus made visible to the software being built during the compilation step.<br><br>Default initialization for <code>CFLAGS</code> varies depending on what is being built: <ul style="list-style-type: none"> <li><code>TARGET_CFLAGS</code> when building for the target</li> <li><code>BUILD_CFLAGS</code> when building for the build host (i.e. <code>-native</code>)</li> <li><code>BUILDSDK_CFLAGS</code> when building for an SDK (i.e. <code>nativesdk-</code>)</li> </ul> |
| <code>CLASSOVERRIDE</code> | An internal variable specifying the special class override that should currently apply (e.g. "class-target", "class-native", and so forth). The classes that use this variable (e.g. <code>native</code> , <code>nativesdk</code> , and so forth) set the variable to appropriate values.                                                                                                                                                                                                                                                                                                 |

### Note

`CLASSOVERRIDE` gets its default "class-target" value from the `bitbake.conf` file.

As an example, the following override allows you to install extra files, but only when building for the target:

```
do_install_append_class-target() {
 install my-extra-file ${D}${sysconfdir}
}
```

Here is an example where `FOO` is set to "native" when building for the build host, and to "other" when not building for the build host:

```
FOO_class-native = "native"
FOO = "other"
```

The underlying mechanism behind `CLASSOVERRIDE` is simply that it is included in the default value of [OVERRIDES](#).

|                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|---------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>CLEANBROKEN</code>        | If set to "1" within a recipe, <code>CLEANBROKEN</code> specifies that the <code>make clean</code> command does not work for the software being built. Consequently, the OpenEmbedded build system will not try to run <code>make clean</code> during the <code>do_configure</code> task, which is the default behavior.                                                                                                                                                                                                                                                         |
| <code>COMBINED_FEATURES</code>  | Provides a list of hardware features that are enabled in both <a href="#">MACHINE_FEATURES</a> and <a href="#">DISTRO_FEATURES</a> . This select list of features contains features that make sense to be controlled both at the machine and distribution configuration level. For example, the "bluetooth" feature requires hardware support but should also be optional at the distribution level, in case the hardware supports Bluetooth but you do not ever intend to use it.                                                                                               |
| <code>COMMON_LICENSE_DIR</code> | Points to <code>meta/files/common-licenses</code> in the <a href="#">Source Directory</a> , which is where generic license files reside.                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <code>COMPATIBLE_HOST</code>    | A regular expression that resolves to one or more hosts (when the recipe is native) or one or more targets (when the recipe is non-native) with which a recipe is compatible. The regular expression is matched against <a href="#">HOST_SYS</a> . You can use the variable to stop recipes from being built for classes of systems with which the recipes are not compatible. Stopping these builds is particularly useful with kernels. The variable also helps to increase parsing speed since the build system skips parsing recipes not compatible with the current system. |
| <code>COMPATIBLE_MACHINE</code> | A regular expression that resolves to one or more target machines with which a recipe is compatible. The regular expression is matched against <a href="#">MACHINEOVERRIDES</a> . You can use the variable to stop recipes from being built for machines with which the recipes are not compatible. Stopping these builds is particularly useful with kernels. The variable also helps to increase parsing speed since the build system skips parsing recipes not compatible with the current machine.                                                                           |
| <code>COMPLEMENTARY_GLOB</code> | Defines wildcards to match when installing a list of complementary packages for all the packages explicitly (or implicitly) installed in an image.                                                                                                                                                                                                                                                                                                                                                                                                                               |



## Note

The `COMPLEMENTARY_GLOB` variable uses Unix filename pattern matching ([fnmatch](#)), which is similar to the Unix style pathname pattern expansion ([glob](#)).

The resulting list of complementary packages is associated with an item that can be added to `IMAGE_FEATURES`. An example usage of this is the "dev-pkgs" item that when added to `IMAGE_FEATURES` will install -dev packages (containing headers and other development files) for every package in the image.

To add a new feature item pointing to a wildcard, use a variable flag to specify the feature item name and use the value to specify the wildcard. Here is an example:

```
COMPLEMENTARY_GLOB[dev-pkgs] = '*-dev'
```

### COMPONENTS\_DIR¶

Stores sysroot components for each recipe. The OpenEmbedded build system uses `COMPONENTS_DIR` when constructing recipe-specific sysroots for other recipes.

The default is "`${STAGING_DIR}-components`." (i.e. "`${TMPDIR}/sysroots-components`").

### CONF\_VERSION¶

Tracks the version of the local configuration file (i.e. `local.conf`). The value for `CONF_VERSION` increments each time `build/conf/compatibility` changes.

### CONFFILES¶

Identifies editable or configurable files that are part of a package. If the Package Management System (PMS) is being used to update packages on the target system, it is possible that configuration files you have changed after the original installation and that you now want to remain unchanged are overwritten. In other words, editable files might exist in the package that you do not want reset as part of the package update process. You can use the `CONFFILES` variable to list the files in the package that you wish to prevent the PMS from overwriting during this update process.

To use the `CONFFILES` variable, provide a package name override that identifies the resulting package. Then, provide a space-separated list of files. Here is an example:

```
CONFFILES_${PN} += "${sysconfdir}/file1 \
${sysconfdir}/file2 ${sysconfdir}/file3"
```

A relationship exists between the `CONFFILES` and `FILES` variables. The files listed within `CONFFILES` must be a subset of the files listed within `FILES`. Because the configuration files you provide with `CONFFILES` are simply being identified so that the PMS will not overwrite them, it makes sense that the files must already be included as part of the package through the `FILES` variable.

## Note

When specifying paths as part of the `CONFFILES` variable, it is good practice to use appropriate path variables. For example, `${sysconfdir}` rather than `/etc` or `${bindir}` rather than `/usr/bin`. You can find a list of these variables at the top of the `meta/conf/bitbake.conf` file in the [Source Directory](#).

### CONFIG\_INITRAMFS\_SOURCE¶

Identifies the initial RAM filesystem (initramfs) source files. The OpenEmbedded build system receives and uses this kernel Kconfig variable as an environment variable. By default, the variable is set to null ("").

The `CONFIG_INITRAMFS_SOURCE` can be either a single cpio archive with a `.cpio` suffix or a space-separated list of directories and files for building the initramfs image. A cpio archive should contain a filesystem archive to be used as an initramfs image. Directories should contain a filesystem layout to be included in the initramfs image. Files should contain entries according to the format described by the `usr/gen_init_cpio` program in the kernel tree.

If you specify multiple directories and files, the initramfs image will be the aggregate of all of them.

For information on creating an initramfs, see the "Building an Initial RAM Filesystem (initramfs) Image" section in the Yocto Project Development Tasks Manual.

### CONFIG\_SITE¶

A list of files that contains `autoconf` test results relevant to the current build. This variable is used by the Autotools utilities when running `configure`.

### CONFIGURE\_FLAGS¶

The minimal arguments for GNU configure.

### CONFLICT\_DISTRO\_FEATURES¶

When inheriting the `distro_features_check` class, this variable identifies distribution features that would be in conflict should the recipe be built. In other words, if the `CONFLICT_DISTRO_FEATURES` variable lists a feature that

also appears in `DISTRO_FEATURES` within the current configuration, an error occurs and the build stops.

#### `COPYLEFT_LICENSE_EXCLUDE`

A space-separated list of licenses to exclude from the source archived by the `archiver` class. In other words, if a license in a recipe's `LICENSE` value is in the value of `COPYLEFT_LICENSE_EXCLUDE`, then its source is not archived by the class.

### Note

The `COPYLEFT_LICENSE_EXCLUDE` variable takes precedence over the `COPYLEFT_LICENSE_INCLUDE` variable.

The default value, which is "CLOSED Proprietary", for `COPYLEFT_LICENSE_EXCLUDE` is set by the `copyleft_filter` class, which is inherited by the `archiver` class.

#### `COPYLEFT_LICENSE_INCLUDE`

A space-separated list of licenses to include in the source archived by the `archiver` class. In other words, if a license in a recipe's `LICENSE` value is in the value of `COPYLEFT_LICENSE_INCLUDE`, then its source is archived by the class.

The default value is set by the `copyleft_filter` class, which is inherited by the `archiver` class. The default value includes "GPL\*", "LGPL\*", and "AGPL\*".

#### `COPYLEFT_PN_EXCLUDE`

A list of recipes to exclude in the source archived by the `archiver` class. The `COPYLEFT_PN_EXCLUDE` variable overrides the license inclusion and exclusion caused through the `COPYLEFT_LICENSE_INCLUDE` and `COPYLEFT_LICENSE_EXCLUDE` variables, respectively.

The default value, which is "" indicating to not explicitly exclude any recipes by name, for `COPYLEFT_PN_EXCLUDE` is set by the `copyleft_filter` class, which is inherited by the `archiver` class.

#### `COPYLEFT_PN_INCLUDE`

A list of recipes to include in the source archived by the `archiver` class. The `COPYLEFT_PN_INCLUDE` variable overrides the license inclusion and exclusion caused through the `COPYLEFT_LICENSE_INCLUDE` and `COPYLEFT_LICENSE_EXCLUDE` variables, respectively.

The default value, which is "" indicating to not explicitly include any recipes by name, for `COPYLEFT_PN_INCLUDE` is set by the `copyleft_filter` class, which is inherited by the `archiver` class.

#### `COPYLEFT_RECIPE_TYPES`

A space-separated list of recipe types to include in the source archived by the `archiver` class. Recipe types are `target`, `native`, `nativesdk`, `cross`, `crosssdk`, and `cross-canadian`.

The default value, which is "target\*", for `COPYLEFT_RECIPE_TYPES` is set by the `copyleft_filter` class, which is inherited by the `archiver` class.

#### `COPY_LIC_DIRS`

If set to "1" along with the `COPY_LIC_MANIFEST` variable, the OpenEmbedded build system copies into the image the license files, which are located in `/usr/share/common-licenses`, for each package. The license files are placed in directories within the image itself during build time.

### Note

The `COPY_LIC_DIRS` does not offer a path for adding licenses for newly installed packages to an image, which might be most suitable for read-only filesystems that cannot be upgraded. See the `LICENSE_CREATE_PACKAGE` variable for additional information. You can also reference the "Providing License Text" section in the Yocto Project Development Tasks Manual for information on providing license text.

#### `COPY_LIC_MANIFEST`

If set to "1", the OpenEmbedded build system copies the license manifest for the image to `/usr/share/common-licenses/license.manifest` within the image itself during build time.

### Note

The `COPY_LIC_MANIFEST` does not offer a path for adding licenses for newly installed packages to an image, which might be most suitable for read-only filesystems that cannot be upgraded. See the `LICENSE_CREATE_PACKAGE` variable for additional information. You can also reference the "Providing License Text" section in the Yocto Project Development Tasks Manual for information on providing license text.

`CORE_IMAGE_EXTRA_INSTALL`

Specifies the list of packages to be added to the image. You should only set this variable in the `local.conf` configuration file found in the [Build Directory](#).

This variable replaces `POKY_EXTRA_INSTALL`, which is no longer supported.

`COREBASE`

Specifies the parent directory of the OpenEmbedded-Core Metadata layer (i.e. `meta`).

It is an important distinction that `COREBASE` points to the parent of this layer and not the layer itself. Consider an example where you have cloned the Poky Git repository and retained the `poky` name for your local copy of the repository. In this case, `COREBASE` points to the `poky` folder because it is the parent directory of the `poky/meta` layer.

`COREBASE_FILES`

Lists files from the `COREBASE` directory that should be copied other than the layers listed in the `bblayers.conf` file. The `COREBASE_FILES` variable exists for the purpose of copying metadata from the OpenEmbedded build system into the extensible SDK.

Explicitly listing files in `COREBASE` is needed because it typically contains build directories and other files that should not normally be copied into the extensible SDK. Consequently, the value of `COREBASE_FILES` is used in order to only copy the files that are actually needed.

`CPP`

The minimal command and arguments used to run the C preprocessor.

`CPPFLAGS`

Specifies the flags to pass to the C pre-processor (i.e. to both the C and the C++ compilers). This variable is exported to an environment variable and thus made visible to the software being built during the compilation step.

Default initialization for `CPPFLAGS` varies depending on what is being built:

- `TARGET_CPPFLAGS` when building for the target
- `BUILD_CPPFLAGS` when building for the build host (i.e. `-native`)
- `BUILDSDK_CPPFLAGS` when building for an SDK (i.e. `nativesdk-`)

`CROSS_COMPILE`

The toolchain binary prefix for the target tools. The `CROSS_COMPILE` variable is the same as the `TARGET_PREFIX` variable.

## Note

The OpenEmbedded build system sets the `CROSS_COMPILE` variable only in certain contexts (e.g. when building for kernel and kernel module recipes).

`CVSDIR`

The directory in which files checked out under the CVS system are stored.

`CXX`

The minimal command and arguments used to run the C++ compiler.

`CXXFLAGS`

Specifies the flags to pass to the C++ compiler. This variable is exported to an environment variable and thus made visible to the software being built during the compilation step.

Default initialization for `CXXFLAGS` varies depending on what is being built:

- `TARGET_CXXFLAGS` when building for the target
- `BUILD_CXXFLAGS` when building for the build host (i.e. `-native`)
- `BUILDSDK_CXXFLAGS` when building for an SDK (i.e. `nativesdk-`)

## D

`D`

The destination directory. The location in the [Build Directory](#) where components are installed by the `do_install` task. This location defaults to:

```
${WORKDIR}/image
```

## Caution

Tasks that read from or write to this directory should run under [fakeroot](#).

`DATE`

The date the build was started. Dates appear using the year, month, and day (YMD) format (e.g. "20150209" for February 9th, 2015).

`DATETIME`

The date and time on which the current build started. The format is suitable for timestamps.

`DEBIAN_NOAUTONAME`

When the [debian](#) class is inherited, which is the default behavior, `DEBIAN_NOAUTONAME` specifies a particular package should not be renamed

according to Debian library package naming. You must use the package name as an override when you set this variable. Here is an example from the `fontconfig` recipe:

```
DEBIAN_NOAUTONAME_fontconfig-utils = "1"
```

#### DEBIANNAME¶

When the `debian` class is inherited, which is the default behavior, `DEBIANNAME` allows you to override the library name for an individual package. Overriding the library name in these cases is rare. You must use the package name as an override when you set this variable. Here is an example from the `dbus` recipe:

```
DEBIANNAME_${PN} = "dbus-1"
```

#### DEBUG\_BUILD¶

Specifies to build packages with debugging information. This influences the value of the `SELECTED_OPTIMIZATION` variable.

#### DEBUG\_OPTIMIZATION¶

The options to pass in `TARGET_CFLAGS` and `CFLAGS` when compiling a system for debugging. This variable defaults to `"-O -fno-omit-frame-pointer ${DEBUG_FLAGS} -pipe"`.

#### DEFAULT\_PREFERENCE¶

Specifies a weak bias for recipe selection priority.

The most common usage of this variable is to set it to `"-1"` within a recipe for a development version of a piece of software. Using the variable in this way causes the stable version of the recipe to build by default in the absence of `PREFERRED_VERSION` being used to build the development version.

### Note

The bias provided by `DEFAULT_PREFERENCE` is weak and is overridden by `BBFILE_PRIORITY` if that variable is different between two layers that contain different versions of the same recipe.

#### DEFAULTTUNE¶

The default CPU and Application Binary Interface (ABI) tunings (i.e. the "tune") used by the OpenEmbedded build system. The `DEFAULTTUNE` helps define `TUNE_FEATURES`.

The default tune is either implicitly or explicitly set by the machine (`MACHINE`). However, you can override the setting using available tunes as defined with `AVAILTUNES`.

#### DEPENDS¶

Lists a recipe's build-time dependencies. These are dependencies on other recipes whose contents (e.g. headers and shared libraries) are needed by the recipe at build time.

As an example, consider a recipe `foo` that contains the following assignment:

```
DEPENDS = "bar"
```

The practical effect of the previous assignment is that all files installed by `bar` will be available in the appropriate staging sysroot, given by the `STAGING_DIR*` variables, by the time the `do_configure` task for `foo` runs. This mechanism is implemented by having `do_configure` depend on the `do_populate_sysroot` task of each recipe listed in `DEPENDS`, through a `[deptask]` declaration in the `base` class.

### Note

It seldom is necessary to reference, for example, `STAGING_DIR_HOST` explicitly. The standard classes and build-related variables are configured to automatically use the appropriate staging sysroots.

As another example, `DEPENDS` can also be used to add utilities that run on the build machine during the build. For example, a recipe that makes use of a code generator built by the recipe `codegen` might have the following:

```
DEPENDS = "codegen-native"
```

For more information, see the `native` class and the `EXTRANATIVEPATH` variable.

### Notes

- `DEPENDS` is a list of recipe names. Or, to be more precise, it is a list of `PROVIDES` names, which usually match recipe names. Putting a package name such as `"foo-dev"` in `DEPENDS` does not make sense.

Use "foo" instead, as this will put files from all the packages that make up `foo`, which includes those from `foo-dev`, into the sysroot.

- One recipe having another recipe in `DEPENDS` does not by itself add any runtime dependencies between the packages produced by the two recipes. However, as explained in the "[Automatically Added Runtime Dependencies](#)" section in the Yocto Project Overview and Concepts Manual, runtime dependencies will often be added automatically, meaning `DEPENDS` alone is sufficient for most recipes.
- Counterintuitively, `DEPENDS` is often necessary even for recipes that install precompiled components. For example, if `libfoo` is a precompiled library that links against `libbar`, then linking against `libfoo` requires both `libfoo` and `libbar` to be available in the sysroot. Without a `DEPENDS` from the recipe that installs `libfoo` to the recipe that installs `libbar`, other recipes might fail to link against `libfoo`.

For information on runtime dependencies, see the [RDEPENDS](#) variable. You can also see the "[Tasks](#)" and "[Dependencies](#)" sections in the BitBake User Manual for additional information on tasks and dependencies.

#### DEPLOY\_DIR¶

Points to the general area that the OpenEmbedded build system uses to place images, packages, SDKs, and other output files that are ready to be used outside of the build system. By default, this directory resides within the [Build Directory](#) as `${TMPDIR}/deploy`.

For more information on the structure of the Build Directory, see "[The Build Directory - build/](#)" section. For more detail on the contents of the `deploy` directory, see the "[Images](#)", "[Package Feeds](#)", and "[Application Development SDK](#)" sections all in the Yocto Project Overview and Concepts Manual.

#### DEPLOY\_DIR\_DEB¶

Points to the area that the OpenEmbedded build system uses to place Debian packages that are ready to be used outside of the build system. This variable applies only when [PACKAGE\\_CLASSES](#) contains "package\_deb".

The BitBake configuration file initially defines the `DEPLOY_DIR_DEB` variable as a sub-folder of [DEPLOY\\_DIR](#):

```
DEPLOY_DIR_DEB = "${DEPLOY_DIR}/deb"
```

The [package\\_deb](#) class uses the `DEPLOY_DIR_DEB` variable to make sure the [do\\_package\\_write\\_deb](#) task writes Debian packages into the appropriate folder. For more information on how packaging works, see the "[Package Feeds](#)" section in the Yocto Project Overview and Concepts Manual.

#### DEPLOY\_DIR\_IMAGE¶

Points to the area that the OpenEmbedded build system uses to place images and other associated output files that are ready to be deployed onto the target machine. The directory is machine-specific as it contains the `${MACHINE}` name. By default, this directory resides within the [Build Directory](#) as `${DEPLOY_DIR}/images/${MACHINE}/`.

For more information on the structure of the Build Directory, see "[The Build Directory - build/](#)" section. For more detail on the contents of the `deploy` directory, see the "[Images](#)" and "[Application Development SDK](#)" sections both in the Yocto Project Overview and Concepts Manual.

#### DEPLOY\_DIR\_IPK¶

Points to the area that the OpenEmbedded build system uses to place IPK packages that are ready to be used outside of the build system. This variable applies only when [PACKAGE\\_CLASSES](#) contains "package\_ipk".

The BitBake configuration file initially defines this variable as a sub-folder of [DEPLOY\\_DIR](#):

```
DEPLOY_DIR_IPK = "${DEPLOY_DIR}/ipk"
```

The [package\\_ipk](#) class uses the `DEPLOY_DIR_IPK` variable to make sure the [do\\_package\\_write\\_ipk](#) task writes IPK packages into the appropriate folder. For more information on how packaging works, see the "[Package Feeds](#)" section in the Yocto Project Overview and Concepts Manual.

#### DEPLOY\_DIR\_RPM¶

Points to the area that the OpenEmbedded build system uses to place RPM packages that are ready to be used outside of the build system. This variable applies only when [PACKAGE\\_CLASSES](#) contains "package\_rpm".

The BitBake configuration file initially defines this variable as a sub-folder of [DEPLOY\\_DIR](#):

```
DEPLOY_DIR_RPM = "${DEPLOY_DIR}/rpm"
```

|                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                       | <p>The <code>package_rpm</code> class uses the <code>DEPLOY_DIR_RPM</code> variable to make sure the <code>do_package_write_rpm</code> task writes RPM packages into the appropriate folder. For more information on how packaging works, see the "<a href="#">Package Feeds</a>" section in the Yocto Project Overview and Concepts Manual.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <code>DEPLOY_DIR_TAR</code>           | <p>Points to the area that the OpenEmbedded build system uses to place tarballs that are ready to be used outside of the build system. This variable applies only when <code>PACKAGE_CLASSES</code> contains "package_tar".</p> <p>The BitBake configuration file initially defines this variable as a sub-folder of <code>DEPLOY_DIR</code>:</p> <pre>DEPLOY_DIR_TAR = "\${DEPLOY_DIR}/tar"</pre> <p>The <code>package_tar</code> class uses the <code>DEPLOY_DIR_TAR</code> variable to make sure the <code>do_package_write_tar</code> task writes TAR packages into the appropriate folder. For more information on how packaging works, see the "<a href="#">Package Feeds</a>" section in the Yocto Project Overview and Concepts Manual.</p>                                                                                                                                                                                                                          |
| <code>DEPLOYDIR</code>                | <p>When inheriting the <code>deploy</code> class, the <code>DEPLOYDIR</code> points to a temporary work area for deployed files that is set in the <code>deploy</code> class as follows:</p> <pre>DEPLOYDIR = "\${WORKDIR}/deploy-\${PN}"</pre> <p>Recipes inheriting the <code>deploy</code> class should copy files to be deployed into <code>DEPLOYDIR</code>, and the class will take care of copying them into <code>DEPLOY_DIR_IMAGE</code> afterwards.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <code>DESCRIPTION</code>              | <p>The package description used by package managers. If not set, <code>DESCRIPTION</code> takes the value of the <code>SUMMARY</code> variable.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <code>DISTRO</code>                   | <p>The short name of the distribution. For information on the long name of the distribution, see the <code>DISTRO_NAME</code> variable.</p> <p>The <code>DISTRO</code> variable corresponds to a distribution configuration file whose root name is the same as the variable's argument and whose filename extension is <code>.conf</code>. For example, the distribution configuration file for the Poky distribution is named <code>poky.conf</code> and resides in the <code>meta-poky/conf/distro</code> directory of the <a href="#">Source Directory</a>.</p> <p>Within that <code>poky.conf</code> file, the <code>DISTRO</code> variable is set as follows:</p> <pre>DISTRO = "poky"</pre> <p>Distribution configuration files are located in a <code>conf/distro</code> directory within the <a href="#">Metadata</a> that contains the distribution configuration. The value for <code>DISTRO</code> must not contain spaces, and is typically all lower-case.</p> |
|                                       | <p><b>Note</b></p> <p>If the <code>DISTRO</code> variable is blank, a set of default configurations are used, which are specified within <code>meta/conf/distro/defaultsetup.conf</code> also in the <a href="#">Source Directory</a>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <code>DISTRO_CODENAME</code>          | <p>Specifies a codename for the distribution being built.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <code>DISTRO_EXTRA_RDEPENDS</code>    | <p>Specifies a list of distro-specific packages to add to all images. This variable takes affect through <code>packagegroup-base</code> so the variable only really applies to the more full-featured images that include <code>packagegroup-base</code>. You can use this variable to keep distro policy out of generic images. As with all other distro variables, you set this variable in the distro <code>.conf</code> file.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <code>DISTRO_EXTRA_RRECOMMENDS</code> | <p>Specifies a list of distro-specific packages to add to all images if the packages exist. The packages might not exist or be empty (e.g. kernel modules). The list of packages are automatically installed but you can remove them.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <code>DISTRO_FEATURES</code>          | <p>The software support you want in your distribution for various features. You define your distribution features in the distribution configuration file.</p> <p>In most cases, the presence or absence of a feature in <code>DISTRO_FEATURES</code> is translated to the appropriate option supplied to the configure script during the <code>do_configure</code> task for recipes that optionally support the feature. For example, specifying "x11" in <code>DISTRO_FEATURES</code>, causes every piece of software built for the target that can optionally support X11 to have its X11 support enabled.</p> <p>Two more examples are Bluetooth and NFS support. For a more complete list of features that ships with the Yocto Project and that you can provide with this variable, see the "<a href="#">Distro Features</a>" section.</p>                                                                                                                              |
| <code>DISTRO_FEATURES_BACKFILL</code> | <p>Features to be added to <code>DISTRO_FEATURES</code> if not also present in <code>DISTRO_FEATURES_BACKFILL_CONSIDERED</code>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

|                                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|--------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                  | <p>This variable is set in the <code>meta/conf/bitbake.conf</code> file. It is not intended to be user-configurable. It is best to just reference the variable to see which distro features are being backfilled for all distro configurations. See the <a href="#">Feature Backfilling</a> section for more information.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <code>DISTRO_FEATURES_BACKFILL_CONSIDERED</code> | <p>Features from <a href="#">DISTRO_FEATURES_BACKFILL</a> that should not be backfilled (i.e. added to <a href="#">DISTRO_FEATURES</a>) during the build. See the <a href="#">"Feature Backfilling"</a> section for more information.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <code>DISTRO_FEATURES_DEFAULT</code>             | <p>A convenience variable that gives you the default list of distro features with the exception of any features specific to the C library (<code>libc</code>).</p> <p>When creating a custom distribution, you might find it useful to be able to reuse the default <a href="#">DISTRO_FEATURES</a> options without the need to write out the full set. Here is an example that uses <code>DISTRO_FEATURES_DEFAULT</code> from a custom distro configuration file:</p> <pre>DISTRO_FEATURES += "\${DISTRO_FEATURES_DEFAULT} myfeature"</pre>                                                                                                                                                                                                                                                                                                                                                                                   |
| <code>DISTRO_FEATURES_FILTER_NATIVE</code>       | <p>Specifies a list of features that if present in the target <a href="#">DISTRO_FEATURES</a> value should be included in <code>DISTRO_FEATURES</code> when building native recipes. This variable is used in addition to the features filtered using the <a href="#">DISTRO_FEATURES_NATIVE</a> variable.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <code>DISTRO_FEATURES_FILTER_NATIVESDK</code>    | <p>Specifies a list of features that if present in the target <a href="#">DISTRO_FEATURES</a> value should be included in <code>DISTRO_FEATURES</code> when building nativesdk recipes. This variable is used in addition to the features filtered using the <a href="#">DISTRO_FEATURES_NATIVESDK</a> variable.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <code>DISTRO_FEATURES_NATIVE</code>              | <p>Specifies a list of features that should be included in <a href="#">DISTRO_FEATURES</a> when building native recipes. This variable is used in addition to the features filtered using the <a href="#">DISTRO_FEATURES_FILTER_NATIVE</a> variable.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <code>DISTRO_FEATURES_NATIVESDK</code>           | <p>Specifies a list of features that should be included in <a href="#">DISTRO_FEATURES</a> when building nativesdk recipes. This variable is used in addition to the features filtered using the <a href="#">DISTRO_FEATURES_FILTER_NATIVESDK</a> variable.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <code>DISTRO_NAME</code>                         | <p>The long name of the distribution. For information on the short name of the distribution, see the <a href="#">DISTRO</a> variable.</p> <p>The <code>DISTRO_NAME</code> variable corresponds to a distribution configuration file whose root name is the same as the variable's argument and whose filename extension is <code>.conf</code>. For example, the distribution configuration file for the Poky distribution is named <code>poky.conf</code> and resides in the <code>meta-poky/conf/distro</code> directory of the <a href="#">Source Directory</a>.</p> <p>Within that <code>poky.conf</code> file, the <code>DISTRO_NAME</code> variable is set as follows:</p> <pre>DISTRO_NAME = "Poky (Yocto Project Reference Distro)"</pre> <p>Distribution configuration files are located in a <code>conf/distro</code> directory within the <a href="#">Metadata</a> that contains the distribution configuration.</p> |

## Note

If the `DISTRO_NAME` variable is blank, a set of default configurations are used, which are specified within `meta/conf/distro/defaultsetup.conf` also in the [Source Directory](#).

|                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>DISTRO_VERSION</code>  | <p>The version of the distribution.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <code>DISTROOVERRIDES</code> | <p>A colon-separated list of overrides specific to the current distribution. By default, this list includes the value of <a href="#">DISTRO</a>.</p> <p>You can extend <code>DISTROOVERRIDES</code> to add extra overrides that should apply to the distribution.</p> <p>The underlying mechanism behind <code>DISTROOVERRIDES</code> is simply that it is included in the default value of <a href="#">OVERRIDES</a>.</p>                                                                                                                                                                                                                                                                                                                                            |
| <code>DL_DIR</code>          | <p>The central download directory used by the build process to store downloads. By default, <code>DL_DIR</code> gets files suitable for mirroring for everything except Git repositories. If you want tarballs of Git repositories, use the <a href="#">BB_GENERATE_MIRROR_TARBALLS</a> variable.</p> <p>You can set this directory by defining the <code>DL_DIR</code> variable in the <code>conf/local.conf</code> file. This directory is self-maintaining and you should not have to touch it. By default, the directory is <code>downloads</code> in the <a href="#">Build Directory</a>.</p> <pre>#DL_DIR += "\${TOPDIR}/downloads"</pre> <p>To specify a different download directory, simply remove the comment from the line and provide your directory.</p> |



During a first build, the system downloads many different source code tarballs from various upstream projects. Downloading can take a while, particularly if your network connection is slow. Tarballs are all stored in the directory defined by `DL_DIR` and the build system looks there first to find source tarballs.

## Note

When wiping and rebuilding, you can preserve this directory to speed up this part of subsequent builds.

You can safely share this directory between multiple builds on the same development machine. For additional information on how the build process gets source files when working behind a firewall or proxy server, see this specific question in the "FAQ" chapter. You can also refer to the "Working Behind a Network Proxy" Wiki page.

`DOC_COMPRESS`

When inheriting the `compress_doc` class, this variable sets the compression policy used when the OpenEmbedded build system compresses man pages and info pages. By default, the compression method used is `gz` (gzip). Other policies available are `xz` and `bz2`.

For information on policies and on how to use this variable, see the comments in the `meta/classes/compress_doc.bbclass` file.

## E

`EFI_PROVIDER`

When building bootable images (i.e. where `hddimg`, `iso`, or `wic.vmdk` is in `IMAGE_FSTYPES`), the `EFI_PROVIDER` variable specifies the EFI bootloader to use. The default is "grub-efi", but "systemd-boot" can be used instead.

See the `systemd-boot` and `image-live` classes for more information.

`ENABLE_BINARY_LOCALE_GENERATION`

Variable that controls which locales for `glibc` are generated during the build (useful if the target device has 64Mbytes of RAM or less).

`ERR_REPORT_DIR`

When used with the `report-error` class, specifies the path used for storing the debug files created by the `error-reporting-tool`, which allows you to submit build errors you encounter to a central database. By default, the value of this variable is `${LOG_DIR}/error-report`.

You can set `ERR_REPORT_DIR` to the path you want the error reporting tool to store the debug files as follows in your `local.conf` file:

```
ERR_REPORT_DIR = "path"
```

`ERROR_QA`

Specifies the quality assurance checks whose failures are reported as errors by the OpenEmbedded build system. You set this variable in your distribution configuration file. For a list of the checks you can control with this variable, see the "`insane.bbclass`" section.

`EXCLUDE_FROM_SHLIBS`

Triggers the OpenEmbedded build system's shared libraries resolver to exclude an entire package when scanning for shared libraries.

## Note

The shared libraries resolver's functionality results in part from the internal function `package_do_shlibs`, which is part of the `do_package` task. You should be aware that the shared libraries resolver might implicitly define some dependencies between packages.

The `EXCLUDE_FROM_SHLIBS` variable is similar to the `PRIVATE_LIBS` variable, which excludes a package's particular libraries only and not the whole package.

Use the `EXCLUDE_FROM_SHLIBS` variable by setting it to "1" for a particular package:

```
EXCLUDE_FROM_SHLIBS = "1"
```

`EXCLUDE_FROM_WORLD`

Directs BitBake to exclude a recipe from world builds (i.e. `bitbake world`). During world builds, BitBake locates, parses and builds all recipes found in every layer exposed in the `bbayers.conf` configuration file.

To exclude a recipe from a world build using this variable, set the variable to "1" in the recipe.

## Note

Recipes added to `EXCLUDE_FROM_WORLD` may still be built during a world build in order to satisfy dependencies



of other recipes. Adding a recipe to `EXCLUDE_FROM_WORLD` only ensures that the recipe is not explicitly added to the list of build targets in a world build.

|                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>EXTENDPE</code>              | <p>Used with file and pathnames to create a prefix for a recipe's version based on the recipe's <code>PE</code> value. If <code>PE</code> is set and greater than zero for a recipe, <code>EXTENDPE</code> becomes that value (e.g if <code>PE</code> is equal to "1" then <code>EXTENDPE</code> becomes "1_"). If a recipe's <code>PE</code> is not set (the default) or is equal to zero, <code>EXTENDPE</code> becomes "".</p> <p>See the <code>STAMP</code> variable for an example.</p>                                                                                                                                                    |
| <code>EXTENDPKGVS</code>           | <p>The full package version specification as it appears on the final packages produced by a recipe. The variable's value is normally used to fix a runtime dependency to the exact same version of another package in the same recipe:</p> <pre>RDEPENDS_\${PN}-additional-module = "\${PN} (= \${EXTENDPKGVS})"</pre> <p>The dependency relationships are intended to force the package manager to upgrade these types of packages in lock-step.</p>                                                                                                                                                                                           |
| <code>EXTERNAL_KERNEL_TOOLS</code> | <p>When set, the <code>EXTERNAL_KERNEL_TOOLS</code> variable indicates that these tools are not in the source tree.</p> <p>When kernel tools are available in the tree, they are preferred over any externally installed tools. Setting the <code>EXTERNAL_KERNEL_TOOLS</code> variable tells the OpenEmbedded build system to prefer the installed external tools. See the <code>kernel-yocto</code> class in <code>meta/classes</code> to see how the variable is used.</p>                                                                                                                                                                   |
| <code>EXTERNALSRC</code>           | <p>When inheriting the <code>externalsrc</code> class, this variable points to the source tree, which is outside of the OpenEmbedded build system. When set, this variable sets the <code>S</code> variable, which is what the OpenEmbedded build system uses to locate unpacked recipe source code.</p> <p>For more information on <code>externalsrc.bbclass</code>, see the "<code>externalsrc.bbclass</code>" section. You can also find information on how to use this variable in the "<code>Building Software from an External Source</code>" section in the Yocto Project Development Tasks Manual.</p>                                  |
| <code>EXTERNALSRC_BUILD</code>     | <p>When inheriting the <code>externalsrc</code> class, this variable points to the directory in which the recipe's source code is built, which is outside of the OpenEmbedded build system. When set, this variable sets the <code>B</code> variable, which is what the OpenEmbedded build system uses to locate the Build Directory.</p> <p>For more information on <code>externalsrc.bbclass</code>, see the "<code>externalsrc.bbclass</code>" section. You can also find information on how to use this variable in the "<code>Building Software from an External Source</code>" section in the Yocto Project Development Tasks Manual.</p> |
| <code>EXTRA_AUTORECONF</code>      | <p>For recipes inheriting the <code>autotools</code> class, you can use <code>EXTRA_AUTORECONF</code> to specify extra options to pass to the <code>autoreconf</code> command that is executed during the <code>do_configure</code> task.</p> <p>The default value is "<code>--exclude=autopoint</code>".</p>                                                                                                                                                                                                                                                                                                                                   |
| <code>EXTRA_IMAGE_FEATURES</code>  | <p>A list of additional features to include in an image. When listing more than one feature, separate them with a space.</p> <p>Typically, you configure this variable in your <code>local.conf</code> file, which is found in the <code>Build Directory</code>. Although you can use this variable from within a recipe, best practices dictate that you do not.</p>                                                                                                                                                                                                                                                                           |

## Note

To enable primary features from within the image recipe, use the `IMAGE_FEATURES` variable.

Here are some examples of features you can add:

- "dbg-pkgs" - Adds -dbg packages for all installed packages including symbol information for debugging and profiling.
- "debug-tweaks" - Makes an image suitable for debugging. For example, allows root logins without passwords and enables post-installation logging. See the 'allow-empty-password' and 'post-install-logging' features in the "`Image Features`" section for more information.
- "dev-pkgs" - Adds -dev packages for all installed packages. This is useful if you want to develop against the libraries in the image.
- "read-only-rootfs" - Creates an image whose root filesystem is read-only. See the

"[Creating a Read-Only Root Filesystem](#)"  
 section in the Yocto Project  
 Development Tasks Manual for  
 more information

"tools-debug" - Adds debugging tools such as gdb and  
 strace.

"tools-sdk" - Adds development tools such as gcc, make,  
 pkgconfig and so forth.

"tools-testapps" - Adds useful testing tools such as  
 ts\_print, aplay, arecord and so  
 forth.

For a complete list of image features that ships with the Yocto Project, see the "[Image Features](#)" section.

For an example that shows how to customize your image by using this variable, see the "[Customizing Images Using Custom IMAGE\\_FEATURES and EXTRA\\_IMAGE\\_FEATURES](#)" section in the Yocto Project Development Tasks Manual.

#### EXTRA\_IMAGECMD¶

Specifies additional options for the image creation command that has been specified in [IMAGE\\_CMD](#). When setting this variable, use an override for the associated image type. Here is an example:

```
EXTRA_IMAGECMD_ext3 ?= "-i 4096"
```

#### EXTRA\_IMAGEDEPENDS¶

A list of recipes to build that do not provide packages for installing into the root filesystem.

Sometimes a recipe is required to build the final image but is not needed in the root filesystem. You can use the [EXTRA\\_IMAGEDEPENDS](#) variable to list these recipes and thus specify the dependencies. A typical example is a required bootloader in a machine configuration.

### Note

To add packages to the root filesystem, see the various  
 \*[RDEPENDS](#) and \*[RRECOMMENDS](#) variables.

#### EXTRANATIVEPATH¶

A list of subdirectories of `${STAGING\_BINDIR\_NATIVE}` added to the beginning of the environment variable `PATH`. As an example, the following prepends "`${STAGING\_BINDIR\_NATIVE}/foo:${STAGING\_BINDIR\_NATIVE}/bar:`" to `PATH`:

```
EXTRANATIVEPATH = "foo bar"
```

#### EXTRA\_OECMAKE¶

Additional [CMake](#) options. See the [cmake](#) class for additional information.

#### EXTRA\_OECONF¶

Additional `configure` script options. See [PACKAGECONFIG\\_CONFGARGS](#) for additional information on passing configure script options.

#### EXTRA\_OEMAKE¶

Additional GNU make options.

Because the [EXTRA\\_OEMAKE](#) defaults to "", you need to set the variable to specify any required GNU options.

[PARALLEL\\_MAKE](#) and [PARALLEL\\_MAKEINST](#) also make use of [EXTRA\\_OEMAKE](#) to pass the required flags.

#### EXTRA\_OESCONS¶

When inheriting the [scons](#) class, this variable specifies additional configuration options you want to pass to the `scons` command line.

#### EXTRA\_USERS\_PARAMS¶

When inheriting the [extrausers](#) class, this variable provides image level user and group operations. This is a more global method of providing user and group configuration as compared to using the [useradd](#) class, which ties user and group configurations to a specific recipe.

The set list of commands you can configure using the [EXTRA\\_USERS\\_PARAMS](#) is shown in the [extrausers](#) class. These commands map to the normal Unix commands of the same names:

```
EXTRA_USERS_PARAMS = "\
useradd -p '' tester; \
groupadd developers; \
userdel nobody; \
groupdel -g video; \
groupmod -g 1020 developers; \
usermod -s /bin/sh tester; \
"
```

## F

#### FEATURE\_PACKAGES¶

Defines one or more packages to include in an image when a specific item is included in `IMAGE_FEATURES`. When setting the value, `FEATURE_PACKAGES` should have the name of the feature item as an override. Here is an example:

```
FEATURE_PACKAGES_widget = "package1 package2"
```

In this example, if "widget" were added to `IMAGE_FEATURES`, `package1` and `package2` would be included in the image.

## Note

Packages installed by features defined through `FEATURE_PACKAGES` are often package groups. While similarly named, you should not confuse the `FEATURE_PACKAGES` variable with package groups, which are discussed elsewhere in the documentation.

### FEED\_DEPLOYDIR\_BASE\_URI¶

Points to the base URL of the server and location within the document-root that provides the metadata and packages required by OPKG to support runtime package management of IPK packages. You set this variable in your `local.conf` file.

Consider the following example:

```
FEED_DEPLOYDIR_BASE_URI = "http://192.168.7.1/BOARD-dir"
```

This example assumes you are serving your packages over HTTP and your databases are located in a directory named `BOARD-dir`, which is underneath your HTTP server's document-root. In this case, the OpenEmbedded build system generates a set of configuration files for you in your target that work with the feed.

### FILES¶

The list of files and directories that are placed in a package. The `PACKAGES` variable lists the packages generated by a recipe.

To use the `FILES` variable, provide a package name override that identifies the resulting package. Then, provide a space-separated list of files or paths that identify the files you want included as part of the resulting package. Here is an example:

```
FILES_${PN} += "${bindir}/mydir1 ${bindir}/mydir2/myfile"
```

## Notes

- When specifying files or paths, you can pattern match using Python's `glob` syntax. For details on the syntax, see the documentation by following the previous link.
- When specifying paths as part of the `FILES` variable, it is good practice to use appropriate path variables. For example, use `${sysconfdir}` rather than `/etc`, or `${bindir}` rather than `/usr/bin`. You can find a list of these variables at the top of the `meta/conf/bitbake.conf` file in the [Source Directory](#). You will also find the default values of the various `FILES_*` variables in this file.

If some of the files you provide with the `FILES` variable are editable and you know they should not be overwritten during the package update process by the Package Management System (PMS), you can identify these files so that the PMS will not overwrite them. See the `CONFFILES` variable for information on how to identify these files to the PMS.

### FILES\_SOLIBSDEV¶

Defines the file specification to match `SOLIBSDEV`. In other words, `FILES_SOLIBSDEV` defines the full path name of the development symbolic link (symlink) for shared libraries on the target platform.

The following statement from the `bitbake.conf` shows how it is set:

```
FILES_SOLIBSDEV ?= "${base_libdir}/lib*${SOLIBSDEV} ${libdir}/lib*${SOLIBSDEV}
```



### FILESEXTRAPATHS¶

Extends the search path the OpenEmbedded build system uses when looking for files and patches as it processes recipes and append files. The default directories BitBake uses when it processes recipes are initially defined by the `FILESPATH` variable. You can extend `FILESPATH` variable by using `FILESEXTRAPATHS`.

Best practices dictate that you accomplish this by using `FILESEXTRAPATHS` from within a `.bbappend` file and that you prepend paths as follows:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
```

In the above example, the build system first looks for files in a directory that has the same name as the corresponding append file.

## Note

When extending `FILESEXTRAPATHS`, be sure to use the immediate expansion (`:=`) operator. Immediate expansion makes sure that BitBake evaluates `THISDIR` at the time the directive is encountered rather than at some later time when expansion might result in a directory that does not contain the files you need.

Also, include the trailing separating colon character if you are prepending. The trailing colon character is necessary because you are directing BitBake to extend the path by prepending directories to the search path.

Here is another common use:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
```

In this example, the build system extends the `FILESPATH` variable to include a directory named `files` that is in the same directory as the corresponding append file.

This next example specifically adds three paths:

```
FILESEXTRAPATHS_prepend := "path_1:path_2:path_3:"
```

A final example shows how you can extend the search path and include a `MACHINE`-specific override, which is useful in a BSP layer:

```
FILESEXTRAPATHS_prepend_intel-x86-common := "${THISDIR}/${PN}:"
```

The previous statement appears in the `linux-yocto-dev.bbappend` file, which is found in the Yocto Project [Source Repositories](#) in `meta-intel/common/recipes-kernel/linux`. Here, the machine override is a special `PACKAGE_ARCH` definition for multiple `meta-intel` machines.

## Note

For a layer that supports a single BSP, the override could just be the value of `MACHINE`.

By prepending paths in `.bbappend` files, you allow multiple append files that reside in different layers but are used for the same recipe to correctly extend the path.

## FILESOVERRIDES¶

A subset of `OVERRIDES` used by the OpenEmbedded build system for creating `FILESPATH`. The `FILESOVERRIDES` variable uses overrides to automatically extend the `FILESPATH` variable. For an example of how that works, see the `FILESPATH` variable description. Additionally, you find more information on how overrides are handled in the "[Conditional Syntax \(Overrides\)](#)" section of the BitBake User Manual.

By default, the `FILESOVERRIDES` variable is defined as:

```
FILESOVERRIDES = "${TRANSLATED_TARGET_ARCH}:${MACHINEOVERRIDES}:${DISTROOVE
```



## Note

Do not hand-edit the `FILESOVERRIDES` variable. The values match up with expected overrides and are used in an expected manner by the build system.

## FILESPATH¶

The default set of directories the OpenEmbedded build system uses when searching for patches and files.

During the build process, BitBake searches each directory in `FILESPATH` in the specified order when looking for files and patches specified by each `file:// URI` in a recipe's `SRC_URI` statements.

The default value for the `FILESPATH` variable is defined in the `base.bbclass` class found in `meta/classes` in the [Source Directory](#):

```
FILESPATH = "${@base_set_filespath(["${FILE_DIRNAME}/${BP}", \
"${FILE_DIRNAME}/${BPN}", "${FILE_DIRNAME}/files"l, d)}
```

The FILESPATH variable is automatically extended using the overrides from the [FILESOVERRIDES](#) variable.

Notes

- Do not hand-edit the FILESPATH variable. If you want the build system to look in directories other than the defaults, extend the FILESPATH variable by using the [FILESEXTRAPATHS](#) variable.
- Be aware that the default FILESPATH directories do not map to directories in custom layers where append files ( `.bbappend`) are used. If you want the build system to find patches or files that reside with your append files, you need to extend the FILESPATH variable by using the [FILESEXTRAPATHS](#) variable.

You can take advantage of this searching behavior in useful ways. For example, consider a case where the following directory structure exists for general and machine-specific configurations:

```
files/defconfig
files/MACHINEA/defconfig
files/MACHINEB/defconfig
```

Also in the example, the SRC\_URI statement contains "file://defconfig". Given this scenario, you can set [MACHINE](#) to "MACHINEA" and cause the build system to use files from files/MACHINEA. Set MACHINE to "MACHINEB" and the build system uses files from files/MACHINEB. Finally, for any machine other than "MACHINEA" and "MACHINEB", the build system uses files from files/defconfig.

You can find out more about the patching process in the "[Patching](#)" section in the Yocto Project Overview and Concepts Manual and the "[Patching Code](#)" section in the Yocto Project Development Tasks Manual. See the [do\\_patch](#) task as well.

FILESYSTEM\_PERMS\_TABLES¶

Allows you to define your own file permissions settings table as part of your configuration for the packaging process. For example, suppose you need a consistent set of custom permissions for a set of groups and users across an entire work project. It is best to do this in the packages themselves but this is not always possible.

By default, the OpenEmbedded build system uses the fs-perms.txt, which is located in the meta/files folder in the [Source Directory](#). If you create your own file permissions setting table, you should place it in your layer or the distro's layer.

You define the FILESYSTEM\_PERMS\_TABLES variable in the conf/local.conf file, which is found in the [Build Directory](#), to point to your custom fs-perms.txt. You can specify more than a single file permissions setting table. The paths you specify to these files must be defined within the [BBPATH](#) variable.

For guidance on how to create your own file permissions settings table file, examine the existing fs-perms.txt.

FONT\_EXTRA\_RDEPENDS¶

When inheriting the [fontcache](#) class, this variable specifies the runtime dependencies for font packages. By default, the FONT\_EXTRA\_RDEPENDS is set to "fontconfig-utils".

FONT\_PACKAGES¶

When inheriting the [fontcache](#) class, this variable identifies packages containing font files that need to be cached by Fontconfig. By default, the fontcache class assumes that fonts are in the recipe's main package (i.e. `_${PN}`). Use this variable if fonts you need are in a package other than that main package.

FORCE\_RO\_REMOVE¶

Forces the removal of the packages listed in ROOTFS\_RO\_UNNEEDED during the generation of the root filesystem.

Set the variable to "1" to force the removal of these packages.

FULL\_OPTIMIZATION¶

The options to pass in [TARGET\\_CFLAGS](#) and [CFLAGS](#) when compiling an optimized system. This variable defaults to "-O2 -pipe \${DEBUG\_FLAGS}".

G

GCCPIE¶

Enables Position Independent Executables (PIE) within the GNU C Compiler (GCC). Enabling PIE in the GCC makes Return Oriented Programming (ROP) attacks much more difficult to execute.

By default the security\_flags.inc file enables PIE by setting the variable as follows:

```
GCCPIE ?= "--enable-default-pie"
```

**GCCVERSION**

Specifies the default version of the GNU C Compiler (GCC) used for compilation. By default, `GCCVERSION` is set to "8.x" in the `meta/conf/distro/include/tcmode-default.inc` include file:

```
GCCVERSION ?= "8.%"
```

You can override this value by setting it in a configuration file such as the `local.conf`.

**GDB**

The minimal command and arguments to run the GNU Debugger.

**GITDIR**

The directory in which a local copy of a Git repository is stored when it is cloned.

**GLIBC\_GENERATE\_LOCALES**

Specifies the list of GLIBC locales to generate should you not wish to generate all LIBC locals, which can be time consuming.

**Note**

If you specifically remove the locale `en_US.UTF-8`, you must set `IMAGE_LINGUAS` appropriately.

You can set `GLIBC_GENERATE_LOCALES` in your `local.conf` file. By default, all locales are generated.

```
GLIBC_GENERATE_LOCALES = "en_GB.UTF-8 en_US.UTF-8"
```

**GROUPADD\_PARAM**

When inheriting the `useradd` class, this variable specifies for a package what parameters should be passed to the `groupadd` command if you wish to add a group to the system when the package is installed.

Here is an example from the `dbus` recipe:

```
GROUPADD_PARAM_${PN} = "-r netdev"
```

For information on the standard Linux shell command `groupadd`, see <http://linux.die.net/man/8/groupadd>.

**GROUPMEMS\_PARAM**

When inheriting the `useradd` class, this variable specifies for a package what parameters should be passed to the `groupmems` command if you wish to modify the members of a group when the package is installed.

For information on the standard Linux shell command `groupmems`, see <http://linux.die.net/man/8/groupmems>.

**GRUB\_GFXSERIAL**

Configures the GNU GRand Unified Bootloader (GRUB) to have graphics and serial in the boot menu. Set this variable to "1" in your `local.conf` or distribution configuration file to enable graphics and serial in the menu.

See the `grub-efi` class for more information on how this variable is used.

**GRUB\_OPTS**

Additional options to add to the GNU GRand Unified Bootloader (GRUB) configuration. Use a semi-colon character ( ; ) to separate multiple options.

The `GRUB_OPTS` variable is optional. See the `grub-efi` class for more information on how this variable is used.

**GRUB\_TIMEOUT**

Specifies the timeout before executing the default `LABEL` in the GNU GRand Unified Bootloader (GRUB).

The `GRUB_TIMEOUT` variable is optional. See the `grub-efi` class for more information on how this variable is used.

**GTKIMMODULES\_PACKAGES**

When inheriting the `gtk-immodules-cache` class, this variable specifies the packages that contain the GTK+ input method modules being installed when the modules are in packages other than the main package.

**H****Homepage**

Website where more information about the software the recipe is building can be found.

**HOST\_ARCH**

The name of the target architecture, which is normally the same as `TARGET_ARCH`. The OpenEmbedded build system supports many architectures. Here is an example list of architectures supported. This list is by no means complete as the architecture is configurable:

```
arm
i586
x86_64
powerpc
powerpc64
mips
```

mipsel

**HOST\_CC\_ARCH**

Specifies architecture-specific compiler flags that are passed to the C compiler.

Default initialization for `HOST_CC_ARCH` varies depending on what is being built:

- `TARGET_CC_ARCH` when building for the target
- `BUILD_CC_ARCH` when building for the build host (i.e. `-native`)
- `BUILDSDK_CC_ARCH` when building for an SDK (i.e. `nativesdk-`)

**HOST\_OS**

Specifies the name of the target operating system, which is normally the same as the `TARGET_OS`. The variable can be set to "linux" for `glibc`-based systems and to "linux-musl" for `musl`. For ARM/EABI targets, there are also "linux-gnueabi" and "linux-musleabi" values possible.

**HOST\_PREFIX**

Specifies the prefix for the cross-compile toolchain. `HOST_PREFIX` is normally the same as `TARGET_PREFIX`.

**HOST\_SYS**

Specifies the system, including the architecture and the operating system, for which the build is occurring in the context of the current recipe.

The OpenEmbedded build system automatically sets this variable based on `HOST_ARCH`, `HOST_VENDOR`, and `HOST_OS` variables.

## Note

You do not need to set the variable yourself.

Consider these two examples:

- Given a native recipe on a 32-bit x86 machine running Linux, the value is "i686-linux".
- Given a recipe being built for a little-endian MIPS target running Linux, the value might be "mipsel-linux".

**HOSTTOOLS**

A space-separated list (filter) of tools on the build host that should be allowed to be called from within build tasks. Using this filter helps reduce the possibility of host contamination. If a tool specified in the value of `HOSTTOOLS` is not found on the build host, the OpenEmbedded build system produces an error and the build is not started.

For additional information, see `HOSTTOOLS_NONFATAL`.

**HOSTTOOLS\_NONFATAL**

A space-separated list (filter) of tools on the build host that should be allowed to be called from within build tasks. Using this filter helps reduce the possibility of host contamination. Unlike `HOSTTOOLS`, the OpenEmbedded build system does not produce an error if a tool specified in the value of `HOSTTOOLS_NONFATAL` is not found on the build host. Thus, you can use `HOSTTOOLS_NONFATAL` to filter optional host tools.

**HOST\_VENDOR**

Specifies the name of the vendor. `HOST_VENDOR` is normally the same as `TARGET_VENDOR`.

I

**ICECC\_DISABLED**

Disables or enables the `icecc` (Icecream) function. For more information on this function and best practices for using this variable, see the "`icecc.bbclass`" section.

Setting this variable to "1" in your `local.conf` disables the function:

```
ICECC_DISABLED ??= "1"
```

To enable the function, set the variable as follows:

```
ICECC_DISABLED = ""
```

**ICECC\_ENV\_EXEC**

Points to the `icecc-create-env` script that you provide. This variable is used by the `icecc` class. You set this variable in your `local.conf` file.

If you do not point to a script that you provide, the OpenEmbedded build system uses the default script provided by the `icecc-create-env.bb` recipe, which is a modified version and not the one that comes with `icecc`.

**ICECC\_PARALLEL\_MAKE**

Extra options passed to the `make` command during the `do_compile` task that specify parallel compilation. This variable usually takes the form of "-j x", where `x` represents the maximum number of parallel threads `make` can run.

## Note

The options passed affect builds on all enabled machines on the network, which are machines running the `iceccd` daemon.

If your enabled machines support multiple cores, coming up with the maximum number of parallel threads that gives you the best performance could take some experimentation since machine speed, network lag, available memory, and existing machine loads can all affect build time. Consequently, unlike the `PARALLEL_MAKE` variable, there is no rule-of-thumb for setting `ICECC_PARALLEL_MAKE` to achieve optimal performance.

If you do not set `ICECC_PARALLEL_MAKE`, the build system does not use it (i.e. the system does not detect and assign the number of cores as is done with `PARALLEL_MAKE`).

`ICECC_PATH`

The location of the `icecc` binary. You can set this variable in your `local.conf` file. If your `local.conf` file does not define this variable, the `icecc` class attempts to define it by locating `icecc` using `which`.

`ICECC_USER_CLASS_BL`

Identifies user classes that you do not want the Icecream distributed compile support to consider. This variable is used by the `icecc` class. You set this variable in your `local.conf` file.

When you list classes using this variable, you are "blacklisting" them from distributed compilation across remote hosts. Any classes you list will be distributed and compiled locally.

`ICECC_USER_PACKAGE_BL`

Identifies user recipes that you do not want the Icecream distributed compile support to consider. This variable is used by the `icecc` class. You set this variable in your `local.conf` file.

When you list packages using this variable, you are "blacklisting" them from distributed compilation across remote hosts. Any packages you list will be distributed and compiled locally.

`ICECC_USER_PACKAGE_WL`

Identifies user recipes that use an empty `PARALLEL_MAKE` variable that you want to force remote distributed compilation on using the Icecream distributed compile support. This variable is used by the `icecc` class. You set this variable in your `local.conf` file.

`IMAGE_BASENAME`

The base name of image output files. This variable defaults to the recipe name (`${PN}`).

`IMAGE_BOOT_FILES`

A space-separated list of files installed into the boot partition when preparing an image using the Wic tool with the `booting-partition` source plugin. By default, the files are installed under the same name as the source files. To change the installed name, separate it from the original name with a semi-colon (;). Source files need to be located in `DEPLOY_DIR_IMAGE`. Here are two examples:

```
IMAGE_BOOT_FILES = "u-boot.img uImage;kernel"
IMAGE_BOOT_FILES = "u-boot.${UBOOT_SUFFIX} ${KERNEL_IMAGETYPE}"
```

Alternatively, source files can be picked up using a glob pattern. In this case, the destination file must have the same name as the base name of the source file path. To install files into a directory within the target location, pass its name after a semi-colon (;). Here are two examples:

```
IMAGE_BOOT_FILES = "bcm2835-bootfiles/*"
IMAGE_BOOT_FILES = "bcm2835-bootfiles/*;boot/"
```

The first example installs all files from `${DEPLOY_DIR_IMAGE}/bcm2835-bootfiles` into the root of the target partition. The second example installs the same files into a `boot` directory within the target partition.

You can find information on how to use the Wic tool in the "[Creating Partitioned Images Using Wic](#)" section of the Yocto Project Development Tasks Manual. Reference material for Wic is located in the "[OpenEmbedded Kickstart \(.wks\) Reference](#)" chapter.

`IMAGE_CLASSES`

A list of classes that all images should inherit. You typically use this variable to specify the list of classes that register the different types of images the OpenEmbedded build system creates.

The default value for `IMAGE_CLASSES` is `image_types`. You can set this variable in your `local.conf` or in a distribution configuration file.

For more information, see `meta/classes/image_types.bbclass` in the [Source Directory](#).

`IMAGE_CMD`

Specifies the command to create the image file for a specific image type, which corresponds to the value set in `IMAGE_FSTYPES`, (e.g. `ext3`, `btrfs`, and so forth). When setting this variable, you should use an override for the associated type. Here is an example:

```
IMAGE_CMD_jffs2 = "mkfs.jffs2 --root=${IMAGE_ROOTFS} \
--faketime --output=${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.jffs2 \
${EXTRA_IMAGECMD}"
```



You typically do not need to set this variable unless you are adding support for a new image type. For more examples on how to set this variable, see the [image\\_types](#) class file, which is `meta/classes/image_types.bbclass`.

#### IMAGE\_DEVICE\_TABLES¶

Specifies one or more files that contain custom device tables that are passed to the `makedevs` command as part of creating an image. These files list basic device nodes that should be created under `/dev` within the image. If `IMAGE_DEVICE_TABLES` is not set, `files/device_table-minimal.txt` is used, which is located by `BBPATH`. For details on how you should write device table files, see `meta/files/device_table-minimal.txt` as an example.

#### IMAGE\_FEATURES¶

The primary list of features to include in an image. Typically, you configure this variable in an image recipe. Although you can use this variable from your `local.conf` file, which is found in the [Build Directory](#), best practices dictate that you do not.

### Note

To enable extra features from outside the image recipe, use the `EXTRA_IMAGE_FEATURES` variable.

For a list of image features that ships with the Yocto Project, see the ["Image Features"](#) section.

For an example that shows how to customize your image by using this variable, see the ["Customizing Images Using Custom IMAGE\\_FEATURES and EXTRA\\_IMAGE\\_FEATURES"](#) section in the Yocto Project Development Tasks Manual.

#### IMAGE\_FSTYPES¶

Specifies the formats the OpenEmbedded build system uses during the build when creating the root filesystem. For example, setting `IMAGE_FSTYPES` as follows causes the build system to create root filesystems using two formats: `.ext3` and `.tar.bz2`:

```
IMAGE_FSTYPES = "ext3 tar.bz2"
```

For the complete list of supported image formats from which you can choose, see [IMAGE\\_TYPES](#).

### Notes

- If an image recipe uses the "inherit image" line and you are setting `IMAGE_FSTYPES` inside the recipe, you must set `IMAGE_FSTYPES` prior to using the "inherit image" line.
- Due to the way the OpenEmbedded build system processes this variable, you cannot update its contents by using `_append` or `_prepend`. You must use the `+=` operator to add one or more options to the `IMAGE_FSTYPES` variable.

#### IMAGE\_INSTALL¶

Used by recipes to specify the packages to install into an image through the `image` class. Use the `IMAGE_INSTALL` variable with care to avoid ordering issues.

Image recipes set `IMAGE_INSTALL` to specify the packages to install into an image through `image.bbclass`. Additionally, "helper" classes such as the `core-image` class exist that can take lists used with [IMAGE\\_FEATURES](#) and turn them into auto-generated entries in `IMAGE_INSTALL` in addition to its default contents.

When you use this variable, it is best to use it as follows:

```
IMAGE_INSTALL_append = " package-name"
```

Be sure to include the space between the quotation character and the start of the package name or names.

### Caution

- When working with a `core-image-minimal-initramfs` image, do not use the `IMAGE_INSTALL` variable to specify packages for installation. Instead, use the `PACKAGE_INSTALL` variable, which allows the initial RAM filesystem

(initramfs) recipe to use a fixed set of packages and not be affected by `IMAGE_INSTALL`. For information on creating an initramfs, see the "[Building an Initial RAM Filesystem \(initramfs\) Image](#)" section in the Yocto Project Development Tasks Manual.

- Using `IMAGE_INSTALL` with the `+=` BitBake operator within the `/conf/local.conf` file or from within an image recipe is not recommended. Use of this operator in these ways can cause ordering issues. Since `core-image.bbclass` sets `IMAGE_INSTALL` to a default value using the `?=` operator, using a `+=` operation against `IMAGE_INSTALL` results in unexpected behavior when used within `conf/local.conf`. Furthermore, the same operation from within an image recipe may or may not succeed depending on the specific situation. In both these cases, the behavior is contrary to how most users expect the `+=` operator to work.

#### `IMAGE_LINGUAS`

Specifies the list of locales to install into the image during the root filesystem construction process. The OpenEmbedded build system automatically splits locale files, which are used for localization, into separate packages. Setting the `IMAGE_LINGUAS` variable ensures that any locale packages that correspond to packages already selected for installation into the image are also installed. Here is an example:

```
IMAGE_LINGUAS = "pt-br de-de"
```

In this example, the build system ensures any Brazilian Portuguese and German locale files that correspond to packages in the image are installed (i.e. `*-locale-pt-br` and `*-locale-de-de` as well as `*-locale-pt` and `*-locale-de`, since some software packages only provide locale files by language and not by country-specific language).

See the `GLIBC_GENERATE_LOCALES` variable for information on generating GLIBC locales.

#### `IMAGE_MANIFEST`

The manifest file for the image. This file lists all the installed packages that make up the image. The file contains package information on a line-per-package basis as follows:

```
packagename packagearch version
```

The `image` class defines the manifest file as follows:

```
IMAGE_MANIFEST = "${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.manifest"
```

The location is derived using the `DEPLOY_DIR_IMAGE` and `IMAGE_NAME` variables. You can find information on how the image is created in the "[Image Generation](#)" section in the Yocto Project Overview and Concepts Manual.

#### `IMAGE_NAME`

The name of the output image files minus the extension. This variable is derived using the `IMAGE_BASENAME`, `MACHINE`, and `DATETIME` variables:

```
IMAGE_NAME = "${IMAGE_BASENAME}-${MACHINE}-${DATETIME}"
```

#### `IMAGE_OVERHEAD_FACTOR`

Defines a multiplier that the build system applies to the initial image size for cases when the multiplier times the returned disk usage value for the image is greater than the sum of `IMAGE_ROOTFS_SIZE` and `IMAGE_ROOTFS_EXTRA_SPACE`. The result of the multiplier applied to the initial image size creates free disk space in the image as overhead. By default, the build process uses a multiplier of 1.3 for this variable. This default value results in 30% free disk space added to the image when this method is used to determine the final generated image size. You should be aware that post install scripts and the package management system uses disk space inside this overhead area. Consequently, the multiplier does not produce an image with all the theoretical free disk space. See `IMAGE_ROOTFS_SIZE` for information on how the build system determines the overall image size.

The default 30% free disk space typically gives the image enough room to boot and allows for basic post installs while still leaving a small amount of free disk space. If 30% free space is inadequate, you can increase the default value. For example, the following setting gives you 50% free space added to the image:

```
IMAGE_OVERHEAD_FACTOR = "1.5"
```

Alternatively, you can ensure a specific amount of free disk space is added to the image by using the `IMAGE_ROOTFS_EXTRA_SPACE` variable.

#### `IMAGE_PKGTYPE`

Defines the package type (i.e. DEB, RPM, IPK, or TAR) used by the OpenEmbedded build system. The variable is defined appropriately by the `package_deb`,

`package_rpm`, `package_ipk`, or `package_tar` class.

## Warning

The `package_tar` class is broken and is not supported. It is recommended that you do not use it.

The `populate_sdk_*` and `image` classes use the `IMAGE_PKGTYPE` for packaging up images and SDKs.

You should not set the `IMAGE_PKGTYPE` manually. Rather, the variable is set indirectly through the appropriate `package_*` class using the `PACKAGE_CLASSES` variable. The OpenEmbedded build system uses the first package type (e.g. DEB, RPM, or IPK) that appears with the variable

## Note

Files using the `.tar` format are never used as a substitute packaging format for DEB, RPM, and IPK formatted files for your image or SDK.

### IMAGE\_POSTPROCESS\_COMMAND¶

Specifies a list of functions to call once the OpenEmbedded build system creates the final image output files. You can specify functions separated by semicolons:

```
IMAGE_POSTPROCESS_COMMAND += "function; ... "
```

If you need to pass the root filesystem path to a command within the function, you can use `${IMAGE_ROOTFS}`, which points to the directory that becomes the root filesystem image. See the `IMAGE_ROOTFS` variable for more information.

### IMAGE\_PREPROCESS\_COMMAND¶

Specifies a list of functions to call before the OpenEmbedded build system creates the final image output files. You can specify functions separated by semicolons:

```
IMAGE_PREPROCESS_COMMAND += "function; ... "
```

If you need to pass the root filesystem path to a command within the function, you can use `${IMAGE_ROOTFS}`, which points to the directory that becomes the root filesystem image. See the `IMAGE_ROOTFS` variable for more information.

### IMAGE\_ROOTFS¶

The location of the root filesystem while it is under construction (i.e. during the `do_rootfs` task). This variable is not configurable. Do not change it.

### IMAGE\_ROOTFS\_ALIGNMENT¶

Specifies the alignment for the output image file in Kbytes. If the size of the image is not a multiple of this value, then the size is rounded up to the nearest multiple of the value. The default value is "1". See `IMAGE_ROOTFS_SIZE` for additional information.

### IMAGE\_ROOTFS\_EXTRA\_SPACE¶

Defines additional free disk space created in the image in Kbytes. By default, this variable is set to "0". This free disk space is added to the image after the build system determines the image size as described in `IMAGE_ROOTFS_SIZE`.

This variable is particularly useful when you want to ensure that a specific amount of free disk space is available on a device after an image is installed and running. For example, to be sure 5 Gbytes of free disk space is available, set the variable as follows:

```
IMAGE_ROOTFS_EXTRA_SPACE = "5242880"
```

For example, the Yocto Project Build Appliance specifically requests 40 Gbytes of extra space with the line:

```
IMAGE_ROOTFS_EXTRA_SPACE = "41943040"
```

### IMAGE\_ROOTFS\_SIZE¶

Defines the size in Kbytes for the generated image. The OpenEmbedded build system determines the final size for the generated image using an algorithm that takes into account the initial disk space used for the generated image, a requested size for the image, and requested additional free disk space to be added to the image. Programatically, the build system determines the final size of the generated image as follows:

```
if (image-du * overhead) < rootfs-size:
 internal-rootfs-size = rootfs-size + xspace
else:
 internal-rootfs-size = (image-du * overhead) + xspace
```

where:

```
image-du = Returned value of the du command on
 the image.
```

```
overhead = IMAGE_OVERHEAD_FACTOR
```

```
rootfs-size = IMAGE_ROOTFS_SIZE
```

```
internal-rootfs-size = Initial root filesystem
 size before any modifications.
```

```
xspace = IMAGE_ROOTFS_EXTRA_SPACE
```

See the [IMAGE\\_OVERHEAD\\_FACTOR](#) and [IMAGE\\_ROOTFS\\_EXTRA\\_SPACE](#) variables for related information.

#### IMAGE\_TYPEDEP¶

Specifies a dependency from one image type on another. Here is an example from the [image-live](#) class:

```
IMAGE_TYPEDEP_live = "ext3"
```

In the previous example, the variable ensures that when "live" is listed with the [IMAGE\\_FSTYPES](#) variable, the OpenEmbedded build system produces an `ext3` image first since one of the components of the live image is an `ext3` formatted partition containing the root filesystem.

#### IMAGE\_TYPES¶

Specifies the complete list of supported image types by default:

```
btrfs
container
cpio
cpio.gz
cpio.lz4
cpio.lzma
cpio.xz
cramfs
elf
ext2
ext2.bz2
ext2.gz
ext2.lzma
ext3
ext3.gz
ext4
ext4.gz
f2fs
hddimg
iso
jffs2
jffs2.sum
multiubi
squashfs
squashfs-lz4
squashfs-lzo
squashfs-xz
tar
tar.bz2
tar.gz
tar.lz4
tar.xz
ubi
ubifs
wic
wic.bz2
wic.gz
wic.lzma
```

For more information about these types of images, see `meta/classes/image_types*.bbclass` in the [Source Directory](#).

#### INC\_PR¶

Helps define the recipe revision for recipes that share a common `include` file. You can think of this variable as part of the recipe revision as set from within an include file.

Suppose, for example, you have a set of recipes that are used across several projects. And, within each of those recipes the revision (its `PR` value) is set accordingly. In this case, when the revision of those recipes changes, the burden is on you to find all those recipes and be sure that they get changed to reflect the updated version of the recipe. In this scenario, it can get complicated when recipes that are used in many places and provide common functionality are upgraded to a new revision.

A more efficient way of dealing with this situation is to set the `INC_PR` variable inside the `include` files that the recipes share and then expand the `INC_PR` variable within the recipes to help define the recipe revision.

The following provides an example that shows how to use the `INC_PR` variable given a common `include` file that defines the variable. Once the variable is defined in the `include` file, you can use the variable to set the `PR` values in each recipe. You will notice that when you set a recipe's `PR` you can provide more granular revisioning by appending values to the `INC_PR` variable:

```
recipes-graphics/xorg-font/xorg-font-common.inc:INC_PR = "r2"
recipes-graphics/xorg-font/encodings_1.0.4.bb:PR = "${INC_PR}.1"
recipes-graphics/xorg-font/font-util_1.3.0.bb:PR = "${INC_PR}.0"
recipes-graphics/xorg-font/font-alias_1.0.3.bb:PR = "${INC_PR}.3"
```

The first line of the example establishes the baseline revision to be used for all recipes that use the `include` file. The remaining lines in the example are from individual recipes and show how the `PR` value is set.

#### INCOMPATIBLE\_LICENSE¶

Specifies a space-separated list of license names (as they would appear in [LICENSE](#)) that should be excluded from the build. Recipes that provide no alternatives to listed incompatible licenses are not built. Packages that are individually licensed with the specified incompatible licenses will be deleted.

## Note

This functionality is only regularly tested using the following setting:

```
INCOMPATIBLE_LICENSE = "GPL-3.0 LGPL-3.0 AGPL-3.0"
```

Although you can use other settings, you might be required to remove dependencies on or provide alternatives to components that are required to produce a functional system image.

### INHERIT¶

Causes the named class or classes to be inherited globally. Anonymous functions in the class or classes are not executed for the base configuration and in each individual recipe. The OpenEmbedded build system ignores changes to `INHERIT` in individual recipes.

For more information on `INHERIT`, see the "[INHERIT Configuration Directive](#)" section in the Bitbake User Manual.

### INHERIT\_DISTRO¶

Lists classes that will be inherited at the distribution level. It is unlikely that you want to edit this variable.

The default value of the variable is set as follows in the `meta/conf/distro/defaultsetup.conf` file:

```
INHERIT_DISTRO ?= "debian devshell sstate license"
```

### INHIBIT\_DEFAULT\_DEPS¶

Prevents the default dependencies, namely the C compiler and standard C library (libc), from being added to [DEPENDS](#). This variable is usually used within recipes that do not require any compilation using the C compiler.

Set the variable to "1" to prevent the default dependencies from being added.

### INHIBIT\_PACKAGE\_DEBUG\_SPLIT¶

Prevents the OpenEmbedded build system from splitting out debug information during packaging. By default, the build system splits out debugging information during the `do_package` task. For more information on how debug information is split out, see the [PACKAGE\\_DEBUG\\_SPLIT\\_STYLE](#) variable.

To prevent the build system from splitting out debug information during packaging, set the `INHIBIT_PACKAGE_DEBUG_SPLIT` variable as follows:

```
INHIBIT_PACKAGE_DEBUG_SPLIT = "1"
```

### INHIBIT\_PACKAGE\_STRIP¶

If set to "1", causes the build to not strip binaries in resulting packages and prevents the `-dbg` package from containing the source files.

By default, the OpenEmbedded build system strips binaries and puts the debugging symbols into `${PN}-dbg`. Consequently, you should not set `INHIBIT_PACKAGE_STRIP` when you plan to debug in general.

### INHIBIT\_SYSROOT\_STRIP¶

If set to "1", causes the build to not strip binaries in the resulting sysroot.

By default, the OpenEmbedded build system strips binaries in the resulting sysroot. When you specifically set the `INHIBIT_SYSROOT_STRIP` variable to "1" in your recipe, you inhibit this stripping.

If you want to use this variable, include the [staging](#) class. This class uses a `sys_strip()` function to test for the variable and acts accordingly.

## Note

Use of the `INHIBIT_SYSROOT_STRIP` variable occurs in rare and special circumstances. For example, suppose you are building bare-metal firmware by using an external GCC toolchain. Furthermore, even if the toolchain's binaries are strippable, other files exist that are needed for the build that are not strippable.

### INITRAMFS\_FSTYPES¶

Defines the format for the output image of an initial RAM filesystem (initramfs), which is used during boot. Supported formats are the same as those supported by the [IMAGE\\_FSTYPES](#) variable.

The default value of this variable, which is set in the `meta/conf/bitbake.conf` configuration file in the [Source Directory](#), is "cpio.gz". The Linux kernel's initramfs mechanism, as opposed to the initial RAM filesystem [initrd](#) mechanism, expects an optionally compressed cpio archive.

### INITRAMFS\_IMAGE¶

Specifies the PROVIDES name of an image recipe that is used to build an initial RAM filesystem (initramfs) image. In other words, the `INITRAMFS_IMAGE` variable causes an additional recipe to be built as a dependency to whatever root filesystem recipe you might be using (e.g. `core-image-sato`). The initramfs image recipe you provide should set IMAGE\_FSTYPES to INITRAMFS\_FSTYPES.

An initramfs image provides a temporary root filesystem used for early system initialization (e.g. loading of modules needed to locate and mount the "real" root filesystem).

## Note

See the `meta/recipes-core/images/core-image-minimal-initramfs.bb` recipe in the [Source Directory](#) for an example initramfs recipe. To select this sample recipe as the one built to provide the initramfs image, set `INITRAMFS_IMAGE` to `"core-image-minimal-initramfs"`.

You can also find more information by referencing the `meta-poky/conf/local.conf.sample.extended` configuration file in the Source Directory, the image class, and the kernel class to see how to use the `INITRAMFS_IMAGE` variable.

If `INITRAMFS_IMAGE` is empty, which is the default, then no initramfs image is built.

For more information, you can also see the INITRAMFS\_IMAGE\_BUNDLE variable, which allows the generated image to be bundled inside the kernel image. Additionally, for information on creating an initramfs image, see the "[Building an Initial RAM Filesystem \(initramfs\) Image](#)" section in the Yocto Project Development Tasks Manual.

`INITRAMFS_IMAGE_BUNDLE`

Controls whether or not the image recipe specified by INITRAMFS\_IMAGE is run through an extra pass (do\_bundle\_initramfs) during kernel compilation in order to build a single binary that contains both the kernel image and the initial RAM filesystem (initramfs) image. This makes use of the CONFIG\_INITRAMFS\_SOURCE kernel feature.

## Note

Using an extra compilation pass to bundle the initramfs avoids a circular dependency between the kernel recipe and the initramfs recipe should the initramfs include kernel modules. Should that be the case, the initramfs recipe depends on the kernel for the kernel modules, and the kernel depends on the initramfs recipe since the initramfs is bundled inside the kernel image.

The combined binary is deposited into the `tmp/deploy` directory, which is part of the [Build Directory](#).

Setting the variable to `"1"` in a configuration file causes the OpenEmbedded build system to generate a kernel image with the initramfs specified in `INITRAMFS_IMAGE` bundled within:

```
INITRAMFS_IMAGE_BUNDLE = "1"
```

By default, the kernel class sets this variable to a null string as follows:

```
INITRAMFS_IMAGE_BUNDLE ?= ""
```

## Note

You must set the `INITRAMFS_IMAGE_BUNDLE` variable in a configuration file. You cannot set the variable in a recipe file.

See the local.conf.sample.extended file for additional information. Also, for information on creating an initramfs, see the "[Building an Initial RAM Filesystem \(initramfs\) Image](#)" section in the Yocto Project Development Tasks Manual.

`INITRAMFS_LINK_NAME`

The link name of the initial RAM filesystem image. This variable is set in the `meta/classes/kernel-artifact-names.bbclass` file as follows:

```
INITRAMFS_LINK_NAME ?= "initramfs-${KERNEL_ARTIFACT_LINK_NAME}"
```

The value of the `KERNEL_ARTIFACT_LINK_NAME` variable, which is set in the same file, has the following value:

```
KERNEL_ARTIFACT_LINK_NAME ?= "${MACHINE}"
```

See the [MACHINE](#) variable for additional information.

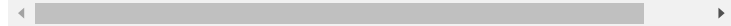
#### INITRAMFS\_NAME¶

The base name of the initial RAM filesystem image. This variable is set in the `meta/classes/kernel-artifact-names.bbclass` file as follows:

```
INITRAMFS_NAME ?= "initramfs-${KERNEL_ARTIFACT_NAME}"
```

The value of the `KERNEL_ARTIFACT_NAME` variable, which is set in the same file, has the following value:

```
KERNEL_ARTIFACT_NAME ?= "${PKGNAME}-${PKGVERSION}-${MACHINE}-${IMAGE_VERSION}"
```



#### INITRD¶

Indicates list of filesystem images to concatenate and use as an initial RAM disk (`initrd`).

The `INITRD` variable is an optional variable used with the [image-live](#) class.

#### INITRD\_IMAGE¶

When building a "live" bootable image (i.e. when `IMAGE_FSTYPES` contains "live"), `INITRD_IMAGE` specifies the image recipe that should be built to provide the initial RAM disk image. The default value is "core-image-minimal-initramfs".

See the [image-live](#) class for more information.

#### INITSCRIPT\_NAME¶

The filename of the initialization script as installed to `${sysconfdir}/init.d`.

This variable is used in recipes when using `update-rc.d.bbclass`. The variable is mandatory.

#### INITSCRIPT\_PACKAGES¶

A list of the packages that contain initscripts. If multiple packages are specified, you need to append the package name to the other `INITSCRIPT_*` as an override.

This variable is used in recipes when using `update-rc.d.bbclass`. The variable is optional and defaults to the `PN` variable.

#### INITSCRIPT\_PARAMS¶

Specifies the options to pass to `update-rc.d`. Here is an example:

```
INITSCRIPT_PARAMS = "start 99 5 2 . stop 20 0 1 6 ."
```

In this example, the script has a runlevel of 99, starts the script in initlevels 2 and 5, and stops the script in levels 0, 1 and 6.

The variable's default value is "defaults", which is set in the `update-rc.d` class.

The value in `INITSCRIPT_PARAMS` is passed through to the `update-rc.d` command. For more information on valid parameters, please see the `update-rc.d` manual page at <http://www.tin.org/bin/man.cgi?section=8&topic=update-rc.d>.

#### INSANE\_SKIP¶

Specifies the QA checks to skip for a specific package within a recipe. For example, to skip the check for symbolic link `.so` files in the main package of a recipe, add the following to the recipe. The package name override must be used, which in this example is `${PN}`:

```
INSANE_SKIP_${PN} += "dev-so"
```

See the "[insane.bbclass](#)" section for a list of the valid QA checks you can specify using this variable.

#### INSTALL\_TIMEZONE\_FILE¶

By default, the `tzdata` recipe packages an `/etc/timezone` file. Set the `INSTALL_TIMEZONE_FILE` variable to "0" at the configuration level to disable this behavior.

#### IPK\_FEED\_URIS¶

When the IPK backend is in use and package management is enabled on the target, you can use this variable to set up `opkg` in the target image to point to package feeds on a nominated server. Once the feed is established, you can perform installations or upgrades using the package manager at runtime.

## K

#### KARCH¶

Defines the kernel architecture used when assembling the configuration. Architectures supported for this release are:

```
powerpc
i386
x86_64
arm
qemu
mips
```

You define the `KARCH` variable in the [BSP Descriptions](#).

|                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| KBRANCH¶              | <p>A regular expression used by the build process to explicitly identify the kernel branch that is validated, patched, and configured during a build. You must set this variable to ensure the exact kernel branch you want is being used by the build process.</p> <p>Values for this variable are set in the kernel's recipe file and the kernel's append file. For example, if you are using the <code>linux-yocto_4.12</code> kernel, the kernel recipe file is the <code>meta/recipes-kernel/linux/linux-yocto_4.12.bb</code> file. <code>KBRANCH</code> is set as follows in that kernel recipe file:</p> <pre>KBRANCH ?= "standard/base"</pre> <p>This variable is also used from the kernel's append file to identify the kernel branch specific to a particular machine or target hardware. Continuing with the previous kernel example, the kernel's append file (i.e. <code>linux-yocto_4.12.bbappend</code>) is located in the BSP layer for a given machine. For example, the append file for the Beaglebone, EdgeRouter, and generic versions of both 32 and 64-bit IA machines (<code>meta-yocto-bsp</code>) is named <code>meta-yocto-bsp/recipes-kernel/linux/linux-yocto_4.12.bbappend</code>. Here are the related statements from that append file:</p> <pre>KBRANCH_genericx86 = "standard/base" KBRANCH_genericx86-64 = "standard/base" KBRANCH_edgerouter = "standard/edgerouter" KBRANCH_beaglebone = "standard/beaglebone" KBRANCH_mpc8315e-rdb = "standard/fsl-mpc8315e-rdb"</pre> <p>The <code>KBRANCH</code> statements identify the kernel branch to use when building for each supported BSP.</p> |
| KBUILD_DEFCONFIG¶     | <p>When used with the <code>kernel-yocto</code> class, specifies an "in-tree" kernel configuration file for use during a kernel build.</p> <p>Typically, when using a <code>defconfig</code> to configure a kernel during a build, you place the file in your layer in the same manner as you would place patch files and configuration fragment files (i.e. "out-of-tree"). However, if you want to use a <code>defconfig</code> file that is part of the kernel tree (i.e. "in-tree"), you can use the <code>KBUILD_DEFCONFIG</code> variable and append the <code>KMACHINE</code> variable to point to the <code>defconfig</code> file.</p> <p>To use the variable, set it in the append file for your kernel recipe using the following form:</p> <pre>KBUILD_DEFCONFIG_KMACHINE ?= defconfig_file</pre> <p>Here is an example from a "raspberrypi2" <code>KMACHINE</code> build that uses a <code>defconfig</code> file named "bcm2709_defconfig":</p> <pre>KBUILD_DEFCONFIG_raspberrypi2 = "bcm2709_defconfig"</pre> <p>As an alternative, you can use the following within your append file:</p> <pre>KBUILD_DEFCONFIG_pn-linux-yocto ?= defconfig_file</pre> <p>For more information on how to use the <code>KBUILD_DEFCONFIG</code> variable, see the "Using an "In-Tree" defconfig File" section in the Yocto Project Linux Kernel Development Manual.</p>                                                                                                                                                                                                                                                            |
| KERNEL_ALT_IMAGETYPE¶ | <p>Specifies an alternate kernel image type for creation in addition to the kernel image type specified using the <code>KERNEL_IMAGETYPE</code> variable.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| KERNEL_ARTIFACT_NAME¶ | <p>Specifies the name of all of the build artifacts. You can change the name of the artifacts by changing the <code>KERNEL_ARTIFACT_NAME</code> variable.</p> <p>The value of <code>KERNEL_ARTIFACT_NAME</code>, which is set in the <code>meta/classes/kernel-artifact-names.bbclass</code> file, has the following default value:</p> <pre>KERNEL_ARTIFACT_NAME ?= "\${PKGGE}-\${PKGCV}-\${PKGR}-\${MACHINE}\${IMAGE_VERSION_</pre> <div><div>◀</div><div></div><div>▶</div></div> <p>See the <code>PKGGE</code>, <code>PKGCV</code>, <code>PKGR</code>, and <code>MACHINE</code> variables for additional information.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |

**Note**

The `IMAGE_VERSION_SUFFIX` variable is set to `DATETIME`.

`KERNEL_CLASSES`¶

A list of classes defining kernel image types that the `kernel` class should inherit. You typically append this variable to enable extended image types. An example is the "kernel-fitimage", which enables fitImage support and resides in `meta/classes/kernel-fitimage.bbclass`. You can register custom kernel image types with the `kernel` class using this variable.



KERNEL\_DEVICETREE¶

Specifies the name of the generated Linux kernel device tree (i.e. the `.dtb`) file.

Note

Legacy support exists for specifying the full path to the device tree. However, providing just the `.dtb` file is preferred.

In order to use this variable, the `kernel-devicetree` class must be inherited.

KERNEL\_DTB\_LINK\_NAME¶

The link name of the kernel device tree binary (DTB). This variable is set in the `meta/classes/kernel-artifact-names.bbclass` file as follows:

```
KERNEL_DTB_LINK_NAME ?= "${KERNEL_ARTIFACT_LINK_NAME}"
```

The value of the `KERNEL_ARTIFACT_LINK_NAME` variable, which is set in the same file, has the following value:

```
KERNEL_ARTIFACT_LINK_NAME ?= "${MACHINE}"
```

See the `MACHINE` variable for additional information.

KERNEL\_DTB\_NAME¶

The base name of the kernel device tree binary (DTB). This variable is set in the `meta/classes/kernel-artifact-names.bbclass` file as follows:

```
KERNEL_DTB_NAME ?= "${KERNEL_ARTIFACT_NAME}"
```

The value of the `KERNEL_ARTIFACT_NAME` variable, which is set in the same file, has the following value:

```
KERNEL_ARTIFACT_NAME ?= "${PKGNAME}-${PKGVER}-${MACHINE}${IMAGE_VERSION_
```



KERNEL\_EXTRA\_ARGS¶

Specifies additional `make` command-line arguments the OpenEmbedded build system passes on when compiling the kernel.

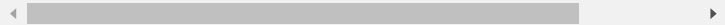
KERNEL\_FEATURES¶

Includes additional kernel metadata. In the OpenEmbedded build system, the default Board Support Packages (BSPs) `Metadata` is provided through the `KMACHINE` and `KBRANCH` variables. You can use the `KERNEL_FEATURES` variable from within the kernel recipe or kernel append file to further add metadata for all BSPs or specific BSPs.

The metadata you add through this variable includes config fragments and features descriptions, which usually includes patches as well as config fragments. You typically override the `KERNEL_FEATURES` variable for a specific machine. In this way, you can provide validated, but optional, sets of kernel configurations and features.

For example, the following example from the `linux-yocto-rt_4.12` kernel recipe adds "netfilter" and "taskstats" features to all BSPs as well as "virtio" configurations to all QEMU machines. The last two statements add specific configurations to targeted machine types:

```
KERNEL_EXTRA_FEATURES ?= "features/netfilter/netfilter.scc features/tasksta
KERNEL_FEATURES_append = " ${KERNEL_EXTRA_FEATURES}"
KERNEL_FEATURES_append_qemuall = " cfg/virtio.scc"
KERNEL_FEATURES_append_qemux86 = " cfg/sound.scc cfg/paravirt_kvm.scc"
KERNEL_FEATURES_append_qemux86-64 = " cfg/sound.scc"
```



KERNEL\_FIT\_LINK\_NAME¶

The link name of the kernel flattened image tree (FIT) image. This variable is set in the `meta/classes/kernel-artifact-names.bbclass` file as follows:

```
KERNEL_FIT_LINK_NAME ?= "${KERNEL_ARTIFACT_LINK_NAME}"
```

The value of the `KERNEL_ARTIFACT_LINK_NAME` variable, which is set in the same file, has the following value:

```
KERNEL_ARTIFACT_LINK_NAME ?= "${MACHINE}"
```

See the `MACHINE` variable for additional information.

KERNEL\_FIT\_NAME¶

The base name of the kernel flattened image tree (FIT) image. This variable is set in the `meta/classes/kernel-artifact-names.bbclass` file as follows:

```
KERNEL_FIT_NAME ?= "${KERNEL_ARTIFACT_NAME}"
```

The value of the `KERNEL_ARTIFACT_NAME` variable, which is set in the same file, has the following value:

```
KERNEL_ARTIFACT_NAME ?= "${PKG}-${PKGVERSION}-${PKGR}-${MACHINE}${IMAGE_VERSION_
```

#### KERNEL\_IMAGE\_LINK\_NAME¶

The link name for the kernel image. This variable is set in the `meta/classes/kernel-artifact-names.bbclass` file as follows:

```
KERNEL_IMAGE_LINK_NAME ?= "${KERNEL_ARTIFACT_LINK_NAME}"
```

The value of the `KERNEL_ARTIFACT_LINK_NAME` variable, which is set in the same file, has the following value:

```
KERNEL_ARTIFACT_LINK_NAME ?= "${MACHINE}"
```

See the [MACHINE](#) variable for additional information.

#### KERNEL\_IMAGE\_MAXSIZE¶

Specifies the maximum size of the kernel image file in kilobytes. If `KERNEL_IMAGE_MAXSIZE` is set, the size of the kernel image file is checked against the set value during the [do\\_sizecheck](#) task. The task fails if the kernel image file is larger than the setting.

`KERNEL_IMAGE_MAXSIZE` is useful for target devices that have a limited amount of space in which the kernel image must be stored.

By default, this variable is not set, which means the size of the kernel image is not checked.

#### KERNEL\_IMAGE\_NAME¶

The base name of the kernel image. This variable is set in the `meta/classes/kernel-artifact-names.bbclass` file as follows:

```
KERNEL_IMAGE_NAME ?= "${KERNEL_ARTIFACT_NAME}"
```

The value of the `KERNEL_ARTIFACT_NAME` variable, which is set in the same file, has the following value:

```
KERNEL_ARTIFACT_NAME ?= "${PKG}-${PKGVERSION}-${PKGR}-${MACHINE}${IMAGE_VERSION_
```

#### KERNEL\_IMAGETYPE¶

The type of kernel to build for a device, usually set by the machine configuration files and defaults to "zImage". This variable is used when building the kernel and is passed to `make` as the target to build.

If you want to build an alternate kernel image type, use the [KERNEL\\_ALT\\_IMAGETYPE](#) variable.

#### KERNEL\_MODULE\_AUTOLOAD¶

Lists kernel modules that need to be auto-loaded during boot.

### Note

This variable replaces the deprecated [module\\_autoload](#) variable.

You can use the `KERNEL_MODULE_AUTOLOAD` variable anywhere that it can be recognized by the kernel recipe or by an out-of-tree kernel module recipe (e.g. a machine configuration file, a distribution configuration file, an append file for the recipe, or the recipe itself).

Specify it as follows:

```
KERNEL_MODULE_AUTOLOAD += "module_name1 module_name2 module_name3"
```

Including `KERNEL_MODULE_AUTOLOAD` causes the OpenEmbedded build system to populate the `/etc/modules-load.d/modname.conf` file with the list of modules to be auto-loaded on boot. The modules appear one-per-line in the file. Here is an example of the most common use case:

```
KERNEL_MODULE_AUTOLOAD += "module_name"
```

For information on how to populate the `modname.conf` file with `modprobe.d` syntax lines, see the [KERNEL\\_MODULE\\_PROBECONF](#) variable.

#### KERNEL\_MODULE\_PROBECONF¶

Provides a list of modules for which the OpenEmbedded build system expects to find `module_conf_modname` values that specify configuration for each of the modules. For information on how to provide those module configurations, see the [module\\_conf\\_\\*](#) variable.

#### KERNEL\_PATH¶

The location of the kernel sources. This variable is set to the value of the [STAGING\\_KERNEL\\_DIR](#) within the `module` class. For information on how this variable is used, see the ["Incorporating Out-of-Tree Modules"](#) section in the Yocto Project Linux Kernel Development Manual.

To help maximize compatibility with out-of-tree drivers used to build modules, the OpenEmbedded build system also recognizes and uses the `KERNEL_SRC` variable, which is identical to the `KERNEL_PATH` variable. Both variables are common variables used by external Makefiles to point to the kernel source directory.

`KERNEL_SRC`

The location of the kernel sources. This variable is set to the value of the `STAGING_KERNEL_DIR` within the `module` class. For information on how this variable is used, see the "[Incorporating Out-of-Tree Modules](#)" section in the Yocto Project Linux Kernel Development Manual.

To help maximize compatibility with out-of-tree drivers used to build modules, the OpenEmbedded build system also recognizes and uses the `KERNEL_PATH` variable, which is identical to the `KERNEL_SRC` variable. Both variables are common variables used by external Makefiles to point to the kernel source directory.

`KERNEL_VERSION`

Specifies the version of the kernel as extracted from `version.h` or `utsrelease.h` within the kernel sources. Effects of setting this variable do not take affect until the kernel has been configured. Consequently, attempting to refer to this variable in contexts prior to configuration will not work.

`KERNELDEPMODDEPEND`

Specifies whether the data referenced through `PKGDATA_DIR` is needed or not. The `KERNELDEPMODDEPEND` does not control whether or not that data exists, but simply whether or not it is used. If you do not need to use the data, set the `KERNELDEPMODDEPEND` variable in your `initramfs` recipe. Setting the variable there when the data is not needed avoids a potential dependency loop.

`KFEATURE_DESCRIPTION`

Provides a short description of a configuration fragment. You use this variable in the `.scf` file that describes a configuration fragment file. Here is the variable used in a file named `smp.scf` to describe SMP being enabled:

```
define KFEATURE_DESCRIPTION "Enable SMP"
```

`KMACHINE`

The machine as known by the kernel. Sometimes the machine name used by the kernel does not match the machine name used by the OpenEmbedded build system. For example, the machine name that the OpenEmbedded build system understands as `core2-32-intel-common` goes by a different name in the Linux Yocto kernel. The kernel understands that machine as `intel-core2-32`. For cases like these, the `KMACHINE` variable maps the kernel machine name to the OpenEmbedded build system machine name.

These mappings between different names occur in the Yocto Linux Kernel's `meta-kernel/linux/linux-yocto_3.19.bbappend` file:

```
LINUX_VERSION_core2-32-intel-common = "3.19.0"
COMPATIBLE_MACHINE_core2-32-intel-common = "${MACHINE}"
SRCREV_meta_core2-32-intel-common = "8897ef68b30e7426bc1d39895e71fb155d6949"
SRCREV_machine_core2-32-intel-common = "43b9eced9ba8a57add36af07736344dcc38"
KMACHINE_core2-32-intel-common = "intel-core2-32"
KBRANCH_core2-32-intel-common = "standard/base"
KERNEL_FEATURES_append_core2-32-intel-common = "${KERNEL_FEATURES_INTEL_COM
```

The `KMACHINE` statement says that the kernel understands the machine name as `intel-core2-32`. However, the OpenEmbedded build system understands the machine as `core2-32-intel-common`.

`KTYPE`

Defines the kernel type to be used in assembling the configuration. The `linux-yocto` recipes define `"standard"`, `"tiny"`, and `"preempt-rt"` kernel types. See the "[Kernel Types](#)" section in the Yocto Project Linux Kernel Development Manual for more information on kernel types.

You define the `KTYPE` variable in the [BSP Descriptions](#). The value you use must match the value used for the `LINUX_KERNEL_TYPE` value used by the kernel recipe.

L

`LABELS`

Provides a list of targets for automatic configuration.

See the `grub-efi` class for more information on how this variable is used.

`LAYERDEPENDS`

Lists the layers, separated by spaces, on which this recipe depends. Optionally, you can specify a specific layer version for a dependency by adding it to the end of the layer name. Here is an example:

```
LAYERDEPENDS_mylayer = "anotherlayer (=3)"
```

In this previous example, version 3 of `"anotherlayer"` is compared against `LAYERVERSION_anotherlayer`.

An error is produced if any dependency is missing or the version numbers (if specified) do not match exactly. This variable is used in the `conf/layer.conf` file and must be suffixed with the name of the specific layer (e.g. `LAYERDEPENDS_mylayer`).

`LAYERDIR`

When used inside the `layer.conf` configuration file, this variable provides the path of the current layer. This variable is not available outside of `layer.conf` and references are expanded immediately when parsing of the file completes.

#### LAYERRECOMMENDS¶

Lists the layers, separated by spaces, recommended for use with this layer.

Optionally, you can specify a specific layer version for a recommendation by adding the version to the end of the layer name. Here is an example:

```
LAYERRECOMMENDS_mylayer = "anotherlayer (=3)"
```

In this previous example, version 3 of "anotherlayer" is compared against `LAYERVERSION_anotherlayer`.

This variable is used in the `conf/layer.conf` file and must be suffixed with the name of the specific layer (e.g. `LAYERRECOMMENDS_mylayer`).

#### LAYERSERIES\_COMPAT¶

Lists the versions of the [OpenEmbedded-Core](#) for which a layer is compatible. Using the `LAYERSERIES_COMPAT` variable allows the layer maintainer to indicate which combinations of the layer and OE-Core can be expected to work. The variable gives the system a way to detect when a layer has not been tested with new releases of OE-Core (e.g. the layer is not maintained).

To specify the OE-Core versions for which a layer is compatible, use this variable in your layer's `conf/layer.conf` configuration file. For the list, use the Yocto Project [Release Name](#) (e.g. zeus). To specify multiple OE-Core versions for the layer, use a space-separated list:

```
LAYERSERIES_COMPAT_Layer_root_name = "zeus warrior"
```

### Note

Setting `LAYERSERIES_COMPAT` is required by the Yocto Project Compatible version 2 standard. The OpenEmbedded build system produces a warning if the variable is not set for any given layer.

See the "[Creating Your Own Layer](#)" section in the Yocto Project Development Tasks Manual.

#### LAYERVERSION¶

Optionally specifies the version of a layer as a single number. You can use this within [LAYERDEPENDS](#) for another layer in order to depend on a specific version of the layer. This variable is used in the `conf/layer.conf` file and must be suffixed with the name of the specific layer (e.g. `LAYERVERSION_mylayer`).

#### LD¶

The minimal command and arguments used to run the linker.

#### LDFLAGS¶

Specifies the flags to pass to the linker. This variable is exported to an environment variable and thus made visible to the software being built during the compilation step.

Default initialization for `LDFLAGS` varies depending on what is being built:

- [TARGET\\_LDFLAGS](#) when building for the target
- [BUILD\\_LDFLAGS](#) when building for the build host (i.e. `-native`)
- [BUILDSDK\\_LDFLAGS](#) when building for an SDK (i.e. `nativesdk-`)

#### LEAD\_SONAME¶

Specifies the lead (or primary) compiled library file (i.e. `.so`) that the [debian](#) class applies its naming policy to given a recipe that packages multiple libraries.

This variable works in conjunction with the `debian` class.

#### LIC\_FILES\_CHKSUM¶

Checksums of the license text in the recipe source code.

This variable tracks changes in license text of the source code files. If the license text is changed, it will trigger a build failure, which gives the developer an opportunity to review any license change.

This variable must be defined for all recipes (unless [LICENSE](#) is set to "CLOSED").

For more information, see the "[Tracking License Changes](#)" section in the Yocto Project Development Tasks Manual.

#### LICENSE¶

The list of source licenses for the recipe. Follow these rules:

- Do not use spaces within individual license names.
- Separate license names using `|` (pipe) when there is a choice between licenses.
- Separate license names using `&` (ampersand) when multiple licenses exist that cover different parts of the source.
- You can use spaces between license names.
- For standard licenses, use the names of the files in `meta/files/common-licenses/` or the [SPDXLICENSEMAP](#) flag names defined in `meta/conf/licenses.conf`.

Here are some examples:

```
LICENSE = "LGPLv2.1 | GPLv3"
LICENSE = "MPL-1 & LGPLv2.1"
LICENSE = "GPLv2+"
```

The first example is from the recipes for Qt, which the user may choose to distribute under either the LGPL version 2.1 or GPL version 3. The second example is from Cairo where two licenses cover different parts of the source code. The final example is from `sysstat`, which presents a single license.

You can also specify licenses on a per-package basis to handle situations where components of the output have different licenses. For example, a piece of software whose code is licensed under GPLv2 but has accompanying documentation licensed under the GNU Free Documentation License 1.2 could be specified as follows:

```
LICENSE = "GFDL-1.2 & GPLv2"
LICENSE_${PN} = "GPLv2"
LICENSE_${PN}-doc = "GFDL-1.2"
```

#### `LICENSE_CREATE_PACKAGE`

Setting `LICENSE_CREATE_PACKAGE` to "1" causes the OpenEmbedded build system to create an extra package (i.e. `${PN}-lic`) for each recipe and to add those packages to the `RRECOMMENDS_${PN}`.

The `${PN}-lic` package installs a directory in `/usr/share/licenses` named `${PN}`, which is the recipe's base name, and installs files in that directory that contain license and copyright information (i.e. copies of the appropriate license files from `meta/common/licenses` that match the licenses specified in the `LICENSE` variable of the recipe metadata and copies of files marked in `LIC_FILES_CHKSUM` as containing license text).

For related information on providing license text, see the `COPY_LIC_DIRS` variable, the `COPY_LIC_MANIFEST` variable, and the "Providing License Text" section in the Yocto Project Development Tasks Manual.

#### `LICENSE_FLAGS`

Specifies additional flags for a recipe you must whitelist through `LICENSE_FLAGS_WHITELIST` in order to allow the recipe to be built. When providing multiple flags, separate them with spaces.

This value is independent of `LICENSE` and is typically used to mark recipes that might require additional licenses in order to be used in a commercial product. For more information, see the "Enabling Commercially Licensed Recipes" section in the Yocto Project Development Tasks Manual.

#### `LICENSE_FLAGS_WHITELIST`

Lists license flags that when specified in `LICENSE_FLAGS` within a recipe should not prevent that recipe from being built. This practice is otherwise known as "whitelisting" license flags. For more information, see the "Enabling Commercially Licensed Recipes" section in the Yocto Project Development Tasks Manual.

#### `LICENSE_PATH`

Path to additional licenses used during the build. By default, the OpenEmbedded build system uses `COMMON_LICENSE_DIR` to define the directory that holds common license text used during the build. The `LICENSE_PATH` variable allows you to extend that location to other areas that have additional licenses:

```
LICENSE_PATH += "path-to-additional-common-licenses"
```

#### `LINUX_KERNEL_TYPE`

Defines the kernel type to be used in assembling the configuration. The linux-yocto recipes define "standard", "tiny", and "preempt-rt" kernel types. See the "Kernel Types" section in the Yocto Project Linux Kernel Development Manual for more information on kernel types.

If you do not specify a `LINUX_KERNEL_TYPE`, it defaults to "standard". Together with `KMACHINE`, the `LINUX_KERNEL_TYPE` variable defines the search arguments used by the kernel tools to find the appropriate description within the kernel `Metadata` with which to build out the sources and configuration.

#### `LINUX_VERSION`

The Linux version from `kernel.org` on which the Linux kernel image being built using the OpenEmbedded build system is based. You define this variable in the kernel recipe. For example, the `linux-yocto-3.4.bb` kernel recipe found in `meta/recipes-kernel/linux` defines the variables as follows:

```
LINUX_VERSION ?= "3.4.24"
```

The `LINUX_VERSION` variable is used to define `PV` for the recipe:

```
PV = "${LINUX_VERSION}+git${SRCPV}"
```

#### `LINUX_VERSION_EXTENSION`

A string extension compiled into the version string of the Linux kernel built with the OpenEmbedded build system. You define this variable in the kernel recipe. For example, the linux-yocto kernel recipes all define the variable as follows:

```
LINUX_VERSION_EXTENSION ?= "-yocto-${LINUX_KERNEL_TYPE}"
```

Defining this variable essentially sets the Linux kernel configuration item `CONFIG_LOCALVERSION`, which is visible through the `uname` command. Here is an example that shows the extension assuming it was set as previously shown:

```
$ uname -r
3.7.0-rc8-custom
```

**LOG\_DIR¶**

Specifies the directory to which the OpenEmbedded build system writes overall log files. The default directory is `${TMPDIR}/log`.

For the directory containing logs specific to each task, see the `T` variable.

**M****MACHINE¶**

Specifies the target device for which the image is built. You define `MACHINE` in the `local.conf` file found in the [Build Directory](#). By default, `MACHINE` is set to "qemux86", which is an x86-based architecture machine to be emulated using QEMU:

```
MACHINE ?= "qemux86"
```

The variable corresponds to a machine configuration file of the same name, through which machine-specific configurations are set. Thus, when `MACHINE` is set to "qemux86" there exists the corresponding `qemux86.conf` machine configuration file, which can be found in the [Source Directory](#) in `meta/conf/machine`.

The list of machines supported by the Yocto Project as shipped include the following:

```
MACHINE ?= "qemuarm"
MACHINE ?= "qemuarm64"
MACHINE ?= "qemumips"
MACHINE ?= "qemumips64"
MACHINE ?= "qemuppc"
MACHINE ?= "qemux86"
MACHINE ?= "qemux86-64"
MACHINE ?= "genericx86"
MACHINE ?= "genericx86-64"
MACHINE ?= "beaglebone"
MACHINE ?= "mpc8315e-rdb"
MACHINE ?= "edgerouter"
```

The last five are Yocto Project reference hardware boards, which are provided in the `meta-yocto-bsp` layer.

**Note**

Adding additional Board Support Package (BSP) layers to your configuration adds new possible settings for `MACHINE`.

**MACHINE\_ARCH¶**

Specifies the name of the machine-specific architecture. This variable is set automatically from `MACHINE` or `TUNE_PKGARCH`. You should not hand-edit the `MACHINE_ARCH` variable.

**MACHINE\_ESSENTIAL\_EXTRA\_RDEPENDS¶**

A list of required machine-specific packages to install as part of the image being built. The build process depends on these packages being present. Furthermore, because this is a "machine-essential" variable, the list of packages are essential for the machine to boot. The impact of this variable affects images based on `packagegroup-core-boot`, including the `core-image-minimal` image.

This variable is similar to the `MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS` variable with the exception that the image being built has a build dependency on the variable's list of packages. In other words, the image will not build if a file in this list is not found.

As an example, suppose the machine for which you are building requires `example-init` to be run during boot to initialize the hardware. In this case, you would use the following in the machine's `.conf` configuration file:

```
MACHINE_ESSENTIAL_EXTRA_RDEPENDS += "example-init"
```

**MACHINE\_ESSENTIAL\_EXTRA\_RRECOMMENDS¶**

A list of recommended machine-specific packages to install as part of the image being built. The build process does not depend on these packages being present. However, because this is a "machine-essential" variable, the list of packages are essential for the machine to boot. The impact of this variable affects images based on `packagegroup-core-boot`, including the `core-image-minimal` image.

This variable is similar to the `MACHINE_ESSENTIAL_EXTRA_RDEPENDS` variable with the exception that the image being built does not have a build dependency on the variable's list of packages. In other words, the image will still build if a package in this list is not found. Typically, this variable is used to handle essential kernel modules, whose functionality may be selected to be built into the kernel rather than as a module, in which case a package will not be produced.

Consider an example where you have a custom kernel where a specific touchscreen driver is required for the machine to be usable. However, the driver can be built as a module or into the kernel depending on the kernel configuration. If the driver is built

as a module, you want it to be installed. But, when the driver is built into the kernel, you still want the build to succeed. This variable sets up a "recommends" relationship so that in the latter case, the build will not fail due to the missing package. To accomplish this, assuming the package for the module was called `kernel-module-ab123`, you would use the following in the machine's `.conf` configuration file:

```
MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS += "kernel-module-ab123"
```

## Note

In this example, the `kernel-module-ab123` recipe needs to explicitly set its `PACKAGES` variable to ensure that BitBake does not use the kernel recipe's `PACKAGES_DYNAMIC` variable to satisfy the dependency.

Some examples of these machine essentials are flash, screen, keyboard, mouse, or touchscreen drivers (depending on the machine).

### MACHINE\_EXTRA\_RDEPENDS¶

A list of machine-specific packages to install as part of the image being built that are not essential for the machine to boot. However, the build process for more fully-featured images depends on the packages being present.

This variable affects all images based on `packagegroup-base`, which does not include the `core-image-minimal` or `core-image-full-cmdline` images.

The variable is similar to the `MACHINE_EXTRA_RRECOMMENDS` variable with the exception that the image being built has a build dependency on the variable's list of packages. In other words, the image will not build if a file in this list is not found.

An example is a machine that has WiFi capability but is not essential for the machine to boot the image. However, if you are building a more fully-featured image, you want to enable the WiFi. The package containing the firmware for the WiFi hardware is always expected to exist, so it is acceptable for the build process to depend upon finding the package. In this case, assuming the package for the firmware was called `wifidriver-firmware`, you would use the following in the `.conf` file for the machine:

```
MACHINE_EXTRA_RDEPENDS += "wifidriver-firmware"
```

### MACHINE\_EXTRA\_RRECOMMENDS¶

A list of machine-specific packages to install as part of the image being built that are not essential for booting the machine. The image being built has no build dependency on this list of packages.

This variable affects only images based on `packagegroup-base`, which does not include the `core-image-minimal` or `core-image-full-cmdline` images.

This variable is similar to the `MACHINE_EXTRA_RDEPENDS` variable with the exception that the image being built does not have a build dependency on the variable's list of packages. In other words, the image will build if a file in this list is not found.

An example is a machine that has WiFi capability but is not essential For the machine to boot the image. However, if you are building a more fully-featured image, you want to enable WiFi. In this case, the package containing the WiFi kernel module will not be produced if the WiFi driver is built into the kernel, in which case you still want the build to succeed instead of failing as a result of the package not being found. To accomplish this, assuming the package for the module was called `kernel-module-examplewifi`, you would use the following in the `.conf` file for the machine:

```
MACHINE_EXTRA_RRECOMMENDS += "kernel-module-examplewifi"
```

### MACHINE\_FEATURES¶

Specifies the list of hardware features the `MACHINE` is capable of supporting. For related information on enabling features, see the `DISTRO_FEATURES`, `COMBINED_FEATURES`, and `IMAGE_FEATURES` variables.

For a list of hardware features supported by the Yocto Project as shipped, see the "Machine Features" section.

### MACHINE\_FEATURES\_BACKFILL¶

Features to be added to `MACHINE_FEATURES` if not also present in `MACHINE_FEATURES_BACKFILL_CONSIDERED`.

This variable is set in the `meta/conf/bitbake.conf` file. It is not intended to be user-configurable. It is best to just reference the variable to see which machine features are being backfilled for all machine configurations. See the "Feature Backfilling" section for more information.

### MACHINE\_FEATURES\_BACKFILL\_CONSIDERED¶

Features from `MACHINE_FEATURES_BACKFILL` that should not be backfilled (i.e. added to `MACHINE_FEATURES`) during the build. See the

**MACHINEOVERRIDES**

"[Feature Backfilling](#)" section for more information.

A colon-separated list of overrides that apply to the current machine. By default, this list includes the value of [MACHINE](#).

You can extend [MACHINEOVERRIDES](#) to add extra overrides that should apply to a machine. For example, all machines emulated in QEMU (e.g. `qemuarm`, `qemux86`, and so forth) include a file named `meta/conf/machine/include/qemu.inc` that prepends the following override to [MACHINEOVERRIDES](#):

```
MACHINEOVERRIDES =. "qemuall:"
```

This override allows variables to be overridden for all machines emulated in QEMU, like in the following example from the `connman-conf` recipe:

```
SRC_URI_append_qemuall = "file://wired.config \
 file://wired-setup \
 "
```

The underlying mechanism behind [MACHINEOVERRIDES](#) is simply that it is included in the default value of [OVERRIDES](#).

**MAINTAINER**

The email address of the distribution maintainer.

**MIRRORS**

Specifies additional paths from which the OpenEmbedded build system gets source code. When the build system searches for source code, it first tries the local download directory. If that location fails, the build system tries locations defined by [PREMIRRORS](#), the upstream source, and then locations specified by [MIRRORS](#) in that order.

Assuming your distribution ([DISTRO](#)) is "poky", the default value for [MIRRORS](#) is defined in the `conf/distro/poky.conf` file in the `meta-poky` Git repository.

**MLPREFIX**

Specifies a prefix has been added to [PN](#) to create a special version of a recipe or package (i.e. a Multilib version). The variable is used in places where the prefix needs to be added to or removed from a the name (e.g. the [BPN](#) variable). [MLPREFIX](#) gets set when a prefix has been added to [PN](#).

**Note**

The "ML" in [MLPREFIX](#) stands for "MultiLib". This representation is historical and comes from a time when `nativesdk` was a suffix rather than a prefix on the recipe name. When `nativesdk` was turned into a prefix, it made sense to set [MLPREFIX](#) for it as well.

To help understand when [MLPREFIX](#) might be needed, consider when [BBCLASSEXTEND](#) is used to provide a `nativesdk` version of a recipe in addition to the target version. If that recipe declares build-time dependencies on tasks in other recipes by using [DEPENDS](#), then a dependency on "foo" will automatically get rewritten to a dependency on "nativesdk-foo". However, dependencies like the following will not get rewritten automatically:

```
do_foo[depends] += "recipe:do_foo"
```

If you want such a dependency to also get transformed, you can do the following:

```
do_foo[depends] += "${MLPREFIX}recipe:do_foo"
```

**module\_autoload**

This variable has been replaced by the [KERNEL\\_MODULE\\_AUTOLOAD](#) variable. You should replace all occurrences of `module_autoload` with additions to [KERNEL\\_MODULE\\_AUTOLOAD](#), for example:

```
module_autoload_rfcomm = "rfcomm"
```

should now be replaced with:

```
KERNEL_MODULE_AUTOLOAD += "rfcomm"
```

See the [KERNEL\\_MODULE\\_AUTOLOAD](#) variable for more information.

**module\_conf**

Specifies `modprobe.d` syntax lines for inclusion in the `/etc/modprobe.d/modname.conf` file.

You can use this variable anywhere that it can be recognized by the kernel recipe or out-of-tree kernel module recipe (e.g. a machine configuration file, a distribution configuration file, an append file for the recipe, or the recipe itself). If you use this variable, you must also be sure to list the module name in the [KERNEL\\_MODULE\\_AUTOLOAD](#) variable.

Here is the general syntax:



```
module_conf_module_name = "modprobe.d-syntax"
```

You must use the kernel module name override.

Run `man modprobe.d` in the shell to find out more information on the exact syntax you want to provide with `module_conf`.

Including `module_conf` causes the OpenEmbedded build system to populate the `/etc/modprobe.d/modname.conf` file with `modprobe.d` syntax lines. Here is an example that adds the options `arg1` and `arg2` to a module named `mymodule`:

```
module_conf_mymodule = "options mymodule arg1=val1 arg2=val2"
```

For information on how to specify kernel modules to auto-load on boot, see the [KERNEL\\_MODULE\\_AUTOLOAD](#) variable.

`MODULE_TARBALL_DEPLOY`

Controls creation of the `modules-*.tgz` file. Set this variable to "0" to disable creation of this file, which contains all of the kernel modules resulting from a kernel build.

`MODULE_TARBALL_LINK_NAME`

The link name of the kernel module tarball. This variable is set in the `meta/classes/kernel-artifact-names.bbclass` file as follows:

```
MODULE_TARBALL_LINK_NAME ?= "${KERNEL_ARTIFACT_LINK_NAME}"
```

The value of the `KERNEL_ARTIFACT_LINK_NAME` variable, which is set in the same file, has the following value:

```
KERNEL_ARTIFACT_LINK_NAME ?= "${MACHINE}"
```

See the [MACHINE](#) variable for additional information.

`MODULE_TARBALL_NAME`

The base name of the kernel module tarball. This variable is set in the `meta/classes/kernel-artifact-names.bbclass` file as follows:

```
MODULE_TARBALL_NAME ?= "${KERNEL_ARTIFACT_NAME}"
```

The value of the `KERNEL_ARTIFACT_NAME` variable, which is set in the same file, has the following value:

```
KERNEL_ARTIFACT_NAME ?= "${PKGGE}-${PKGCV}-${PKGR}-${MACHINE}${IMAGE_VERSION_
```



`MULTIMACH_TARGET_SYS`

Uniquely identifies the type of the target system for which packages are being built. This variable allows output for different types of target systems to be put into different subdirectories of the same output directory.

The default value of this variable is:

```
${PACKAGE_ARCH}${TARGET_VENDOR}-${TARGET_OS}
```

Some classes (e.g. [cross-canadian](#)) modify the `MULTIMACH_TARGET_SYS` value.

See the [STAMP](#) variable for an example. See the [STAGING\\_DIR\\_TARGET](#) variable for more information.

## N

`NATIVESBSTRING`

A string identifying the host distribution. Strings consist of the host distributor ID followed by the release, as reported by the `lsb_release` tool or as read from `/etc/lsb-release`. For example, when running a build on Ubuntu 12.10, the value is "Ubuntu-12.10". If this information is unable to be determined, the value resolves to "Unknown".

This variable is used by default to isolate native shared state packages for different distributions (e.g. to avoid problems with `glibc` version incompatibilities). Additionally, the variable is checked against [SANITY\\_TESTED\\_DISTROS](#) if that variable is set.

`NM`

The minimal command and arguments to run `nm`.

`NO_GENERIC_LICENSE`

Avoids QA errors when you use a non-common, non-CLOSED license in a recipe. Packages exist, such as the `linux-firmware` package, with many licenses that are not in any way common. Also, new licenses are added occasionally to avoid introducing a lot of common license files, which are only applicable to a specific package. `NO_GENERIC_LICENSE` is used to allow copying a license that does not exist in common licenses.

The following example shows how to add `NO_GENERIC_LICENSE` to a recipe:

```
NO_GENERIC_LICENSE[license_name] = "license_file_in_fetched_source"
```

The following is an example that uses the `LICENSE.Abilis.txt` file as the license from the fetched source:

```
NO_GENERIC_LICENSE[Firmware-Abilis] = "LICENSE.Abilis.txt"
```

#### NO\_RECOMMENDATIONS¶

Prevents installation of all "recommended-only" packages. Recommended-only packages are packages installed only through the `RRECOMMENDS` variable). Setting the `NO_RECOMMENDATIONS` variable to "1" turns this feature on:

```
NO_RECOMMENDATIONS = "1"
```

You can set this variable globally in your `local.conf` file or you can attach it to a specific image recipe by using the recipe name override:

```
NO_RECOMMENDATIONS_pn-target_image = "1"
```

It is important to realize that if you choose to not install packages using this variable and some other packages are dependent on them (i.e. listed in a recipe's `RDEPENDS` variable), the OpenEmbedded build system ignores your request and will install the packages to avoid dependency errors.

### Note

Some recommended packages might be required for certain system functionality, such as kernel modules. It is up to you to add packages with the `IMAGE_INSTALL` variable.

Support for this variable exists only when using the IPK and RPM packaging backend. Support does not exist for DEB.

See the `BAD_RECOMMENDATIONS` and the `PACKAGE_EXCLUDE` variables for related information.

#### NOAUTOPACKAGEDEBUG¶

Disables auto package from splitting `.debug` files. If a recipe requires `FILES_${PN}-dbg` to be set manually, the `NOAUTOPACKAGEDEBUG` can be defined allowing you to define the content of the debug package. For example:

```
NOAUTOPACKAGEDEBUG = "1"
FILES_${PN}-dev = "${includedir}/${QT_DIR_NAME}/Qt/*"
FILES_${PN}-dbg = "/usr/src/debug/"
FILES_${QT_BASE_NAME}-demos-doc = "${docdir}/${QT_DIR_NAME}/qch/qt.qch"
```

## O

#### OBJCOPY¶

The minimal command and arguments to run `objcopy`.

#### OBJDUMP¶

The minimal command and arguments to run `objdump`.

#### OE\_BINCONFIG\_EXTRA\_MANGLE¶

When inheriting the `binconfig` class, this variable specifies additional arguments passed to the "sed" command. The sed command alters any paths in configuration scripts that have been set up during compilation. Inheriting this class results in all paths in these scripts being changed to point into the `sysroots/` directory so that all builds that use the script will use the correct directories for the cross compiling layout.

See the `meta/classes/binconfig.bbclass` in the [Source Directory](#) for details on how this class applies these additional sed command arguments. For general information on the `binconfig` class, see the "[binconfig.bbclass](#)" section.

#### OE\_IMPORTS¶

An internal variable used to tell the OpenEmbedded build system what Python modules to import for every Python function run by the system.

### Note

Do not set this variable. It is for internal use only.

#### OE\_INIT\_ENV\_SCRIPT¶

The name of the build environment setup script for the purposes of setting up the environment within the extensible SDK. The default value is "oe-init-build-env".

If you use a custom script to set up your build environment, set the `OE_INIT_ENV_SCRIPT` variable to its name.

#### OE\_TERMINAL¶

Controls how the OpenEmbedded build system spawns interactive terminals on the host development system (e.g. using the BitBake command with the `-C devshell` command-line option). For more information, see the "[Using a Development Shell](#)" section in the Yocto Project Development Tasks Manual.

You can use the following values for the `OE_TERMINAL` variable:

```
auto
gnome
```

```
xfce
rxvt
screen
konsole
none
```

**OEROOT**

The directory from which the top-level build environment setup script is sourced. The Yocto Project provides a top-level build environment setup script: `oe-init-build-env`. When you run this script, the OEROOT variable resolves to the directory that contains the script.

For additional information on how this variable is used, see the initialization script.

**OLDEST\_KERNEL**

Declares the oldest version of the Linux kernel that the produced binaries must support. This variable is passed into the build of the Embedded GNU C Library (glibc).

The default for this variable comes from the `meta/conf/bitbake.conf` configuration file. You can override this default by setting the variable in a custom distribution configuration file.

**OVERRIDES**

A colon-separated list of overrides that currently apply. Overrides are a BitBake mechanism that allows variables to be selectively overridden at the end of parsing. The set of overrides in OVERRIDES represents the "state" during building, which includes the current recipe being built, the machine for which it is being built, and so forth.

As an example, if the string "an-override" appears as an element in the colon-separated list in OVERRIDES, then the following assignment will override `FOO` with the value "overridden" at the end of parsing:

```
FOO_an-override = "overridden"
```

See the "[Conditional Syntax \(Overrides\)](#)" section in the BitBake User Manual for more information on the overrides mechanism.

The default value of OVERRIDES includes the values of the `CLASSOVERRIDE`, `MACHINEOVERRIDES`, and `DISTROOVERRIDES` variables. Another important override included by default is `pn-${PN}`. This override allows variables to be set for a single recipe within configuration (`.conf`) files. Here is an example:

```
FOO_pn-myrecipe = "myrecipe-specific value"
```

**Tip**

An easy way to see what overrides apply is to search for OVERRIDES in the output of the `bitbake -e` command. See the "[Viewing Variable Values](#)" section in the Yocto Project Development Tasks Manual for more information.

**P****P**

The recipe name and version. `P` is comprised of the following:

```
${PN}-${PV}
```

**PACKAGE\_ARCH**

The architecture of the resulting package or packages.

By default, the value of this variable is set to `TUNE_PKGARCH` when building for the target, `BUILD_ARCH` when building for the build host, and `"${SDK_ARCH}-${SDKPKGSUFFIX}"` when building for the SDK.

**Note**

See `SDK_ARCH` for more information.

However, if your recipe's output packages are built specific to the target machine rather than generally for the architecture of the machine, you should set `PACKAGE_ARCH` to the value of `MACHINE_ARCH` in the recipe as follows:

```
PACKAGE_ARCH = "${MACHINE_ARCH}"
```

**PACKAGE\_ARCHS**

Specifies a list of architectures compatible with the target machine. This variable is set automatically and should not normally be hand-edited. Entries are separated using spaces and listed in order of priority. The default value for `PACKAGE_ARCHS` is "all any noarch `${PACKAGE_EXTRA_ARCHS}` `${MACHINE_ARCH}`".

**PACKAGE\_BEFORE\_PN**

Enables easily adding packages to `PACKAGES` before `${PN}` so that those added packages can pick up files that would normally be included in the default package.

**PACKAGE\_CLASSES**

This variable, which is set in the `local.conf` configuration file found in the `conf` folder of the [Build Directory](#), specifies the package manager the OpenEmbedded build system uses when packaging data.

You can provide one or more of the following arguments for the variable:

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

## Warning

While it is a legal option, the `package_tar` class has limited functionality due to no support for package dependencies by that backend. Therefore, it is recommended that you do not use it.

The build system uses only the first argument in the list as the package manager when creating your image or SDK. However, packages will be created using any additional packaging classes you specify. For example, if you use the following in your `local.conf` file:

```
PACKAGE_CLASSES ?= "package_ipk"
```

The OpenEmbedded build system uses the IPK package manager to create your image or SDK.

For information on packaging and build performance effects as a result of the package manager in use, see the "[package.bbclass](#)" section.

**PACKAGE\_DEBUG\_SPLIT\_STYLE**

Determines how to split up the binary and debug information when creating `*-dbg` packages to be used with the GNU Project Debugger (GDB).

With the `PACKAGE_DEBUG_SPLIT_STYLE` variable, you can control where debug information, which can include or exclude source files, is stored:

- `".debug"`: Debug symbol files are placed next to the binary in a `.debug` directory on the target. For example, if a binary is installed into `/bin`, the corresponding debug symbol files are installed in `/bin/.debug`. Source files are placed in `/usr/src/debug`.
- `"debug-file-directory"`: Debug symbol files are placed under `/usr/lib/debug` on the target, and separated by the path from where the binary is installed. For example, if a binary is installed in `/bin`, the corresponding debug symbols are installed in `/usr/lib/debug/bin`. Source files are placed in `/usr/src/debug`.
- `"debug-without-src"`: The same behavior as `".debug"` previously described with the exception that no source files are installed.
- `"debug-with-srcpkg"`: The same behavior as `".debug"` previously described with the exception that all source files are placed in a separate `*-src` pkg. This is the default behavior.

You can find out more about debugging using GDB by reading the "[Debugging With the GNU Project Debugger \(GDB\) Remotely](#)" section in the Yocto Project Development Tasks Manual.

**PACKAGE\_EXCLUDE\_COMPLEMENTARY**

Prevents specific packages from being installed when you are installing complementary packages.

You might find that you want to prevent installing certain packages when you are installing complementary packages. For example, if you are using `IMAGE_FEATURES` to install `dev-pkgs`, you might not want to install all packages from a particular multilib. If you find yourself in this situation, you can use the `PACKAGE_EXCLUDE_COMPLEMENTARY` variable to specify regular expressions to match the packages you want to exclude.

**PACKAGE\_EXCLUDE**

Lists packages that should not be installed into an image. For example:

```
PACKAGE_EXCLUDE = "package_name package_name package_name ..."
```

You can set this variable globally in your `local.conf` file or you can attach it to a specific image recipe by using the recipe name override:

```
PACKAGE_EXCLUDE_pn-target_image = "package_name"
```

If you choose to not install a package using this variable and some other package is dependent on it (i.e. listed in a recipe's `RDEPENDS` variable), the OpenEmbedded build system generates a fatal installation error. Because the build system halts the process with a fatal error, you can use the variable with an iterative development process to remove specific components from a system.

Support for this variable exists only when using the IPK and RPM packaging backend. Support does not exist for DEB.

See the [NO\\_RECOMMENDATIONS](#) and the [BAD\\_RECOMMENDATIONS](#) variables for related information.

**PACKAGE\_EXTRA\_ARCHS¶**

Specifies the list of architectures compatible with the device CPU. This variable is useful when you build for several different devices that use miscellaneous processors such as XScale and ARM926-EJS.

**PACKAGE\_FEED\_ARCHS¶**

Optionally specifies the package architectures used as part of the package feed URIs during the build. When used, the `PACKAGE_FEED_ARCHS` variable is appended to the final package feed URI, which is constructed using the `PACKAGE_FEED_URI` and `PACKAGE_FEED_BASE_PATHS` variables.

**Tip**

You can use the `PACKAGE_FEEDS_ARCHS` variable to whitelist specific package architectures. If you do not need to whitelist specific architectures, which is a common case, you can omit this variable. Omitting the variable results in all available architectures for the current machine being included into remote package feeds.

Consider the following example where the `PACKAGE_FEED_URI`, `PACKAGE_FEED_BASE_PATHS`, and `PACKAGE_FEED_ARCHS` variables are defined in your `local.conf` file:

```
PACKAGE_FEED_URI = "https://example.com/packagerepos/release \
https://example.com/packagerepos/updates"
PACKAGE_FEED_BASE_PATHS = "rpm rpm-dev"
PACKAGE_FEED_ARCHS = "all core2-64"
```

Given these settings, the resulting package feeds are as follows:

```
https://example.com/packagerepos/release/rpm/all
https://example.com/packagerepos/release/rpm/core2-64
https://example.com/packagerepos/release/rpm-dev/all
https://example.com/packagerepos/release/rpm-dev/core2-64
https://example.com/packagerepos/updates/rpm/all
https://example.com/packagerepos/updates/rpm/core2-64
https://example.com/packagerepos/updates/rpm-dev/all
https://example.com/packagerepos/updates/rpm-dev/core2-64
```

**PACKAGE\_FEED\_BASE\_PATHS¶**

Specifies the base path used when constructing package feed URIs. The `PACKAGE_FEED_BASE_PATHS` variable makes up the middle portion of a package feed URI used by the OpenEmbedded build system. The base path lies between the `PACKAGE_FEED_URI` and `PACKAGE_FEED_ARCHS` variables.

Consider the following example where the `PACKAGE_FEED_URI`, `PACKAGE_FEED_BASE_PATHS`, and `PACKAGE_FEED_ARCHS` variables are defined in your `local.conf` file:

```
PACKAGE_FEED_URI = "https://example.com/packagerepos/release \
https://example.com/packagerepos/updates"
PACKAGE_FEED_BASE_PATHS = "rpm rpm-dev"
PACKAGE_FEED_ARCHS = "all core2-64"
```

Given these settings, the resulting package feeds are as follows:

```
https://example.com/packagerepos/release/rpm/all
https://example.com/packagerepos/release/rpm/core2-64
https://example.com/packagerepos/release/rpm-dev/all
https://example.com/packagerepos/release/rpm-dev/core2-64
https://example.com/packagerepos/updates/rpm/all
https://example.com/packagerepos/updates/rpm/core2-64
https://example.com/packagerepos/updates/rpm-dev/all
https://example.com/packagerepos/updates/rpm-dev/core2-64
```

**PACKAGE\_FEED\_URI¶**

Specifies the front portion of the package feed URI used by the OpenEmbedded build system. Each final package feed URI is comprised of `PACKAGE_FEED_URI`, `PACKAGE_FEED_BASE_PATHS`, and `PACKAGE_FEED_ARCHS` variables.

Consider the following example where the `PACKAGE_FEED_URI`, `PACKAGE_FEED_BASE_PATHS`, and `PACKAGE_FEED_ARCHS` variables are defined in your `local.conf` file:

```
PACKAGE_FEED_URI = "https://example.com/packagerepos/release \
https://example.com/packagerepos/updates"
PACKAGE_FEED_BASE_PATHS = "rpm rpm-dev"
PACKAGE_FEED_ARCHS = "all core2-64"
```

Given these settings, the resulting package feeds are as follows:

```
https://example.com/packagerepos/release/rpm/all
https://example.com/packagerepos/release/rpm/core2-64
https://example.com/packagerepos/release/rpm-dev/all
https://example.com/packagerepos/release/rpm-dev/core2-64
https://example.com/packagerepos/updates/rpm/all
https://example.com/packagerepos/updates/rpm/core2-64
https://example.com/packagerepos/updates/rpm-dev/all
https://example.com/packagerepos/updates/rpm-dev/core2-64
```

|                                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>PACKAGE_GROUP</code>               | <p>The <code>PACKAGE_GROUP</code> variable has been renamed to <code>FEATURE_PACKAGES</code>. See the variable description for <code>FEATURE_PACKAGES</code> for information.</p> <p>If if you use the <code>PACKAGE_GROUP</code> variable, the OpenEmbedded build system issues a warning message.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <code>PACKAGE_INSTALL</code>             | <p>The final list of packages passed to the package manager for installation into the image.</p> <p>Because the package manager controls actual installation of all packages, the list of packages passed using <code>PACKAGE_INSTALL</code> is not the final list of packages that are actually installed. This variable is internal to the image construction code. Consequently, in general, you should use the <code>IMAGE_INSTALL</code> variable to specify packages for installation. The exception to this is when working with the <code>core-image-minimal-initramfs</code> image. When working with an initial RAM filesystem (initramfs) image, use the <code>PACKAGE_INSTALL</code> variable. For information on creating an initramfs, see the "<a href="#">Building an Initial RAM Filesystem (initramfs) Image</a>" section in the Yocto Project Development Tasks Manual.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <code>PACKAGE_INSTALL_ATTEMPTONLY</code> | <p>Specifies a list of packages the OpenEmbedded build system attempts to install when creating an image. If a listed package fails to install, the build system does not generate an error. This variable is generally not user-defined.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <code>PACKAGE_PREPROCESS_FUNCS</code>    | <p>Specifies a list of functions run to pre-process the <code>PKGD</code> directory prior to splitting the files out to individual packages.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <code>PACKAGE_WRITE_DEPS</code>          | <p>Specifies a list of dependencies for post-installation and pre-installation scripts on native/cross tools. If your post-installation or pre-installation script can execute at rootfs creation time rather than on the target but depends on a native tool in order to execute, you need to list the tools in <code>PACKAGE_WRITE_DEPS</code>.</p> <p>For information on running post-installation scripts, see the "<a href="#">Post-Installation Scripts</a>" section in the Yocto Project Development Tasks Manual.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <code>PACKAGECONFIG</code>               | <p>This variable provides a means of enabling or disabling features of a recipe on a per-recipe basis. <code>PACKAGECONFIG</code> blocks are defined in recipes when you specify features and then arguments that define feature behaviors. Here is the basic block structure:</p> <pre> PACKAGECONFIG ??= "f1 f2 f3 ..." PACKAGECONFIG[f1] = "--with-f1,--without-f1,build-deps-f1,rt-deps-f1,rt-rec PACKAGECONFIG[f2] = "--with-f2,--without-f2,build-deps-f2,rt-deps-f2,rt-rec PACKAGECONFIG[f3] = "--with-f3,--without-f3,build-deps-f3,rt-deps-f3,rt-rec </pre>  <p>The <code>PACKAGECONFIG</code> variable itself specifies a space-separated list of the features to enable. Following the features, you can determine the behavior of each feature by providing up to five order-dependent arguments, which are separated by commas. You can omit any argument you like but must retain the separating commas. The order is important and specifies the following:</p> <ol style="list-style-type: none"> <li>1. Extra arguments that should be added to the configure script argument list (<code>EXTRA_OECONF</code> or <code>PACKAGECONFIG_CONFARGS</code>) if the feature is enabled.</li> <li>2. Extra arguments that should be added to <code>EXTRA_OECONF</code> or <code>PACKAGECONFIG_CONFARGS</code> if the feature is disabled.</li> <li>3. Additional build dependencies (<code>DEPENDS</code>) that should be added if the feature is enabled.</li> <li>4. Additional runtime dependencies (<code>RDEPENDS</code>) that should be added if the feature is enabled.</li> <li>5. Additional runtime recommendations (<code>RRECOMMENDS</code>) that should be added if the feature is enabled.</li> </ol> <p>Consider the following <code>PACKAGECONFIG</code> block taken from the <code>librsvg</code> recipe. In this example the feature is <code>croco</code>, which has three arguments that determine the feature's behavior.</p> <pre> PACKAGECONFIG ??= "croco" PACKAGECONFIG[croco] = "--with-croco,--without-croco,libcroco" </pre> <p>The <code>--with-croco</code> and <code>libcroco</code> arguments apply only if the feature is enabled. In this case, <code>--with-croco</code> is added to the configure script argument list and <code>libcroco</code> is added to <code>DEPENDS</code>. On the other hand, if the feature is disabled say through a <code>.bbappend</code> file in another layer, then the second argument <code>--without-croco</code> is added to the configure script rather than <code>--with-croco</code>.</p> <p>The basic <code>PACKAGECONFIG</code> structure previously described holds true regardless of whether you are creating a block or changing a block. When creating a block, use the structure inside your recipe.</p> <p>If you want to change an existing <code>PACKAGECONFIG</code> block, you can do so one of two ways:</p> |

- **Append file:** Create an append file named `recipeName.bbappend` in your layer and override the value of `PACKAGECONFIG`. You can either completely override the variable:

```
PACKAGECONFIG = "f4 f5"
```

Or, you can just append the variable:

```
PACKAGECONFIG_append = " f4"
```

- **Configuration file:** This method is identical to changing the block through an append file except you edit your `local.conf` or `mydistro.conf` file. As with append files previously described, you can either completely override the variable:

```
PACKAGECONFIG_pn-recipeName = "f4 f5"
```

Or, you can just amend the variable:

```
PACKAGECONFIG_append_pn-recipeName = " f4"
```

#### PACKAGECONFIG\_CONFARGS¶

A space-separated list of configuration options generated from the `PACKAGECONFIG` setting.

Classes such as `autotools` and `cmake` use `PACKAGECONFIG_CONFARGS` to pass `PACKAGECONFIG` options to `configure` and `cmake`, respectively. If you are using `PACKAGECONFIG` but not a class that handles the `do_configure` task, then you need to use `PACKAGECONFIG_CONFARGS` appropriately.

#### PACKAGEGROUP\_DISABLE\_COMPLEMENTARY¶

For recipes inheriting the `packagegroup` class, setting `PACKAGEGROUP_DISABLE_COMPLEMENTARY` to "1" specifies that the normal complementary packages (i.e. `-dev`, `-dbg`, and so forth) should not be automatically created by the `packagegroup` recipe, which is the default behavior.

#### PACKAGES¶

The list of packages the recipe creates. The default value is the following:

```
${PN}-dbg ${PN}-staticdev ${PN}-dev ${PN}-doc ${PN}-locale ${PACKAGE_BEFORE
```



During packaging, the `do_package` task goes through `PACKAGES` and uses the `FILES` variable corresponding to each package to assign files to the package. If a file matches the `FILES` variable for more than one package in `PACKAGES`, it will be assigned to the earliest (leftmost) package.

Packages in the variable's list that are empty (i.e. where none of the patterns in `FILES_pkg` match any files installed by the `do_install` task) are not generated, unless generation is forced through the `ALLOW_EMPTY` variable.

#### PACKAGES\_DYNAMIC¶

A promise that your recipe satisfies runtime dependencies for optional modules that are found in other recipes. `PACKAGES_DYNAMIC` does not actually satisfy the dependencies, it only states that they should be satisfied. For example, if a hard, runtime dependency (`RDEPENDS`) of another package is satisfied at build time through the `PACKAGES_DYNAMIC` variable, but a package with the module name is never actually produced, then the other package will be broken. Thus, if you attempt to include that package in an image, you will get a dependency failure from the packaging system during the `do_rootfs` task.

Typically, if there is a chance that such a situation can occur and the package that is not created is valid without the dependency being satisfied, then you should use `RRECOMMENDS` (a soft runtime dependency) instead of `RDEPENDS`.

For an example of how to use the `PACKAGES_DYNAMIC` variable when you are splitting packages, see the "[Handling Optional Module Packaging](#)" section in the Yocto Project Development Tasks Manual.

#### PACKAGESPLITFUNCS¶

Specifies a list of functions run to perform additional splitting of files into individual packages. Recipes can either prepend to this variable or prepend to the `populate_packages` function in order to perform additional package splitting. In either case, the function should set `PACKAGES`, `FILES`, `RDEPENDS` and other packaging variables appropriately in order to perform the desired splitting.

#### PARALLEL\_MAKE¶

Extra options passed to the `make` command during the `do_compile` task in order to specify parallel compilation on the local build host. This variable is usually in the form `"-j x"`, where `x` represents the maximum number of parallel threads `make` can run.

### Caution

In order for `PARALLEL_MAKE` to be effective, `make` must be called with `${EXTRA_OEMAKE}`. An easy way to ensure this is to use the `oe_runmake` function.

By default, the OpenEmbedded build system automatically sets this variable to be equal to the number of cores the build system uses.

## Note

If the software being built experiences dependency issues during the `do_compile` task that result in race conditions, you can clear the `PARALLEL_MAKE` variable within the recipe as a workaround. For information on addressing race conditions, see the "[Debugging Parallel Make Races](#)" section in the Yocto Project Development Tasks Manual.

For single socket systems (i.e. one CPU), you should not have to override this variable to gain optimal parallelism during builds. However, if you have very large systems that employ multiple physical CPUs, you might want to make sure the `PARALLEL_MAKE` variable is not set higher than `-j 20`.

For more information on speeding up builds, see the "[Speeding Up a Build](#)" section in the Yocto Project Development Tasks Manual.

`PARALLEL_MAKEINST`

Extra options passed to the `make install` command during the `do_install` task in order to specify parallel installation. This variable defaults to the value of `PARALLEL_MAKE`.

## Notes and Cautions

In order for `PARALLEL_MAKEINST` to be effective, `make` must be called with  `${EXTRA_OEMAKE}`. An easy way to ensure this is to use the `oe_runmake` function.

If the software being built experiences dependency issues during the `do_install` task that result in race conditions, you can clear the `PARALLEL_MAKEINST` variable within the recipe as a workaround. For information on addressing race conditions, see the "[Debugging Parallel Make Races](#)" section in the Yocto Project Development Tasks Manual.

`PATCHRESOLVE`

Determines the action to take when a patch fails. You can set this variable to one of two values: `"noop"` and `"user"`.

The default value of `"noop"` causes the build to simply fail when the OpenEmbedded build system cannot successfully apply a patch. Setting the value to `"user"` causes the build system to launch a shell and places you in the right location so that you can manually resolve the conflicts.

Set this variable in your `local.conf` file.

`PATCHTOOL`

Specifies the utility used to apply patches for a recipe during the `do_patch` task. You can specify one of three utilities: `"patch"`, `"quilt"`, or `"git"`. The default utility used is `"quilt"` except for the quilt-native recipe itself. Because the quilt tool is not available at the time quilt-native is being patched, it uses `"patch"`.

If you wish to use an alternative patching tool, set the variable in the recipe using one of the following:

```
PATCHTOOL = "patch"
PATCHTOOL = "quilt"
PATCHTOOL = "git"
```

`PE`

The epoch of the recipe. By default, this variable is unset. The variable is used to make upgrades possible when the versioning scheme changes in some backwards incompatible way.

`PE` is the default value of the `PKG` variable.

`PF`

Specifies the recipe or package name and includes all version and revision numbers (i.e. `glibc-2.13-r20+svnr15508/` and `bash-4.2-r1/`). This variable is comprised of the following:

```
${PN}-${EXTENDPE}${PV}-${PR}
```

`PIXBUF_PACKAGES`

When inheriting the `pixbufcache` class, this variable identifies packages that contain the pixbuf loaders used with `gdk-pixbuf`. By default, the `pixbufcache` class assumes that the loaders are in the recipe's main package (i.e.  `${PN}`). Use this variable if the loaders you need are in a package other than that main package.

`PKG`



The name of the resulting package created by the OpenEmbedded build system.

## Note

When using the `PKG` variable, you must use a package name override.

For example, when the `debian` class renames the output package, it does so by setting `PKG_packageName`.

`PKG_CONFIG_PATH`

The path to `pkg-config` files for the current build context. `pkg-config` reads this variable from the environment.

`PKGDIR`

Points to the destination directory for files to be packaged before they are split into individual packages. This directory defaults to the following:

```
${WORKDIR}/package
```

Do not change this default.

`PKGDATA_DIR`

Points to a shared, global-state directory that holds data generated during the packaging process. During the packaging process, the `do_packagedata` task packages data for each recipe and installs it into this temporary, shared area. This directory defaults to the following, which you should not change:

```
${STAGING_DIR_HOST}/pkgdata
```

For examples of how this data is used, see the "[Automatically Added Runtime Dependencies](#)" section in the Yocto Project Overview and Concepts Manual and the "[Viewing Package Information with oe-pkgdata-util](#)" section in the Yocto Project Development Tasks Manual. For more information on the shared, global-state directory, see [STAGING\\_DIR\\_HOST](#).

`PKGDEST`

Points to the parent directory for files to be packaged after they have been split into individual packages. This directory defaults to the following:

```
${WORKDIR}/packages-split
```

Under this directory, the build system creates directories for each package specified in [PACKAGES](#). Do not change this default.

`PKGDESTWORK`

Points to a temporary work area where the `do_package` task saves package metadata. The `PKGDESTWORK` location defaults to the following:

```
${WORKDIR}/pkgdata
```

Do not change this default.

The `do_packagedata` task copies the package metadata from `PKGDESTWORK` to [PKGDATA\\_DIR](#) to make it available globally.

`PKGE`

The epoch of the package(s) built by the recipe. By default, `PKGE` is set to [PE](#).

`PKGR`

The revision of the package(s) built by the recipe. By default, `PKGR` is set to [PR](#).

`PKGVS`

The version of the package(s) built by the recipe. By default, `PKGVS` is set to [PV](#).

`PN`

This variable can have two separate functions depending on the context: a recipe name or a resulting package name.

`PN` refers to a recipe name in the context of a file used by the OpenEmbedded build system as input to create a package. The name is normally extracted from the recipe file name. For example, if the recipe is named `expat_2.0.1.bb`, then the default value of `PN` will be "expat".

The variable refers to a package name in the context of a file created or produced by the OpenEmbedded build system.

If applicable, the `PN` variable also contains any special suffix or prefix. For example, using `bash` to build packages for the native machine, `PN` is `bash-native`. Using `bash` to build packages for the target and for Multilib, `PN` would be `bash` and `lib64-bash`, respectively.

`PNBLACKLIST`

Lists recipes you do not want the OpenEmbedded build system to build. This variable works in conjunction with the [blacklist](#) class, which is inherited globally.

To prevent a recipe from being built, use the `PNBLACKLIST` variable in your `local.conf` file. Here is an example that prevents `myrecipe` from being built:

```
PNBLACKLIST[myrecipe] = "Not supported by our organization."
```

`POPULATE_SDK_POST_HOST_COMMAND`

Specifies a list of functions to call once the OpenEmbedded build system has created the host part of the SDK. You can specify functions separated by semicolons:

```
POPULATE_SDK_POST_HOST_COMMAND += "function; ... "
```

If you need to pass the SDK path to a command within a function, you can use `${SDK_DIR}`, which points to the parent directory used by the OpenEmbedded build system when creating SDK output. See the [SDK\\_DIR](#) variable for more information.

**POPULATE\_SDK\_POST\_TARGET\_COMMAND¶**

Specifies a list of functions to call once the OpenEmbedded build system has created the target part of the SDK. You can specify functions separated by semicolons:

```
POPULATE_SDK_POST_TARGET_COMMAND += "function; ... "
```

If you need to pass the SDK path to a command within a function, you can use `${SDK_DIR}`, which points to the parent directory used by the OpenEmbedded build system when creating SDK output. See the [SDK\\_DIR](#) variable for more information.

**PR¶**

The revision of the recipe. The default value for this variable is "r0". Subsequent revisions of the recipe conventionally have the values "r1", "r2", and so forth. When [PV](#) increases, [PR](#) is conventionally reset to "r0".

## Note

The OpenEmbedded build system does not need the aid of [PR](#) to know when to rebuild a recipe. The build system uses the task [input checksums](#) along with the [stamp](#) and [shared state cache](#) mechanisms.

The [PR](#) variable primarily becomes significant when a package manager dynamically installs packages on an already built image. In this case, [PR](#), which is the default value of [PKGRR](#), helps the package manager distinguish which package is the most recent one in cases where many packages have the same [PV](#) (i.e. [PKGVR](#)). A component having many packages with the same [PV](#) usually means that the packages all install the same upstream version, but with later ([PR](#)) version packages including packaging fixes.

## Note

[PR](#) does not need to be increased for changes that do not change the package contents or metadata.

Because manually managing [PR](#) can be cumbersome and error-prone, an automated solution exists. See the "[Working With a PR Service](#)" section in the Yocto Project Development Tasks Manual for more information.

**PREFERRED\_PROVIDER¶**

If multiple recipes provide the same item, this variable determines which recipe is preferred and thus provides the item (i.e. the preferred provider). You should always suffix this variable with the name of the provided item. And, you should define the variable using the preferred recipe's name ([PN](#)). Here is a common example:

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
```

In the previous example, multiple recipes are providing "virtual/kernel". The [PREFERRED\\_PROVIDER](#) variable is set with the name ([PN](#)) of the recipe you prefer to provide "virtual/kernel".

Following are more examples:

```
PREFERRED_PROVIDER_virtual/xserver = "xserver-xf86"
PREFERRED_PROVIDER_virtual/libgl ?= "mesa"
```

For more information, see the "[Using Virtual Providers](#)" section in the Yocto Project Development Tasks Manual.

## Note

If you use a `virtual/*` item with [PREFERRED\\_PROVIDER](#), then any recipe that [PROVIDES](#) that item but is not selected (defined) by [PREFERRED\\_PROVIDER](#) is prevented from building, which is usually desirable since this mechanism is designed to select between mutually exclusive alternative providers.

**PREFERRED\_VERSION¶**

If multiple versions of recipes exist, this variable determines which version is given preference. You must always suffix the variable with the [PN](#) you want to select, and you should set the [PV](#) accordingly for precedence.

The `PREFERRED_VERSION` variable supports limited wildcard use through the `"%"` character. You can use the character to match any number of characters, which can be useful when specifying versions that contain long revision numbers that potentially change. Here are two examples:

```
PREFERRED_VERSION_python = "3.4.0"
PREFERRED_VERSION_linux-yocto = "5.0%"
```

## Important

The use of the `"%"` character is limited in that it only works at the end of the string. You cannot use the wildcard character in any other location of the string.

The specified version is matched against `PV`, which does not necessarily match the version part of the recipe's filename. For example, consider two recipes `foo_1.2.bb` and `foo_git.bb` where `foo_git.bb` contains the following assignment:

```
PV = "1.1+git${SRCPV}"
```

In this case, the correct way to select `foo_git.bb` is by using an assignment such as the following:

```
PREFERRED_VERSION_foo = "1.1+git%"
```

Compare that previous example against the following incorrect example, which does not work:

```
PREFERRED_VERSION_foo = "git"
```

Sometimes the `PREFERRED_VERSION` variable can be set by configuration files in a way that is hard to change. You can use `OVERRIDES` to set a machine-specific override. Here is an example:

```
PREFERRED_VERSION_linux-yocto-qemu86 = "5.0%"
```

Although not recommended, worst case, you can also use the "forcevariable" override, which is the strongest override possible. Here is an example:

```
PREFERRED_VERSION_linux-yocto_forcevariable = "5.0%"
```

## Note

The `_forcevariable` override is not handled specially. This override only works because the default value of `OVERRIDES` includes "forcevariable".

### PREMIRRORS¶

Specifies additional paths from which the OpenEmbedded build system gets source code. When the build system searches for source code, it first tries the local download directory. If that location fails, the build system tries locations defined by `PREMIRRORS`, the upstream source, and then locations specified by `MIRRORS` in that order.

Assuming your distribution (`DISTRO`) is "poky", the default value for `PREMIRRORS` is defined in the `conf/distro/poky.conf` file in the `meta-poky` Git repository.

Typically, you could add a specific server for the build system to attempt before any others by adding something like the following to the `local.conf` configuration file in the `Build Directory`:

```
PREMIRRORS_prepend = "\
git://.*.* http://www.yoctoproject.org/sources/ \n \
ftp://.*.* http://www.yoctoproject.org/sources/ \n \
http://.*.* http://www.yoctoproject.org/sources/ \n \
https://.*.* http://www.yoctoproject.org/sources/ \n"
```

These changes cause the build system to intercept Git, FTP, HTTP, and HTTPS requests and direct them to the `http://sources.mirror`. You can use `file://` URLs to point to local directories or network shares as well.

### PRIORITY¶

Indicates the importance of a package.

`PRIORITY` is considered to be part of the distribution policy because the importance of any given recipe depends on the purpose for which the distribution is being produced. Thus, `PRIORITY` is not normally set within recipes.

You can set `PRIORITY` to "required", "standard", "extra", and "optional", which is the default.

### PRIVATE\_LIBS¶

Specifies libraries installed within a recipe that should be ignored by the OpenEmbedded build system's shared library resolver. This variable is typically used when software being built by a recipe has its own private versions of a library normally provided by another recipe. In this case, you would not want the package containing the private libraries to be set as a dependency on other unrelated packages that should instead depend on the package providing the standard version of the library.

Libraries specified in this variable should be specified by their file name. For example, from the Firefox recipe in meta-browser:

```
PRIVATE_LIBS = "libmozjs.so \
 libxpcom.so \
 libnspr4.so \
 libxul.so \
 libmozalloc.so \
 libplc4.so \
 libplds4.so"
```

For more information, see the "[Automatically Added Runtime Dependencies](#)" section in the Yocto Project Overview and Concepts Manual.

## PROVIDES¶

A list of aliases by which a particular recipe can be known. By default, a recipe's own PN is implicitly already in its PROVIDES list. If a recipe uses PROVIDES, the additional aliases are synonyms for the recipe and can be useful satisfying dependencies of other recipes during the build as specified by DEPENDS.

Consider the following example PROVIDES statement from a recipe file `libav_0.8.11.bb`:

```
PROVIDES += "libpostproc"
```

The PROVIDES statement results in the "libav" recipe also being known as "libpostproc".

In addition to providing recipes under alternate names, the PROVIDES mechanism is also used to implement virtual targets. A virtual target is a name that corresponds to some particular functionality (e.g. a Linux kernel). Recipes that provide the functionality in question list the virtual target in PROVIDES. Recipes that depend on the functionality in question can include the virtual target in DEPENDS to leave the choice of provider open.

Conventionally, virtual targets have names on the form "virtual/function" (e.g. "virtual/kernel"). The slash is simply part of the name and has no syntactical significance.

The PREFERRED\_PROVIDER variable is used to select which particular recipe provides a virtual target.

## Note

A corresponding mechanism for virtual runtime dependencies (packages) exists. However, the mechanism does not depend on any special functionality beyond ordinary variable assignments. For example, `VIRTUAL-RUNTIME_dev_manager` refers to the package of the component that manages the `/dev` directory.

Setting the "preferred provider" for runtime dependencies is as simple as using the following assignment in a configuration file:

```
VIRTUAL-RUNTIME_dev_manager = "udev"
```

## PRSERV\_HOST¶

The network based PR service host and port.

The `conf/local.conf.sample.extended` configuration file in the Source Directory shows how the `PRSERV_HOST` variable is set:

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

You must set the variable if you want to automatically start a local PR service. You can set `PRSERV_HOST` to other values to use a remote PR service.

## PTEST\_ENABLED¶

Specifies whether or not Package Test (ptest) functionality is enabled when building a recipe. You should not set this variable directly. Enabling and disabling building Package Tests at build time should be done by adding "ptest" to (or removing it from) DISTRO\_FEATURES.

## PV¶

The version of the recipe. The version is normally extracted from the recipe filename. For example, if the recipe is named `expat_2.0.1.bb`, then the default value of PV will be "2.0.1". PV is generally not overridden within a recipe unless it is

building an unstable (i.e. development) version from a source code repository (e.g. Git or Subversion).

PV is the default value of the `PKG` variable.

#### `PYTHON_ABI`

When used by recipes that inherit the `distutils3`, `setuptools3`, `distutils`, or `setuptools` classes, denotes the Application Binary Interface (ABI) currently in use for Python. By default, the ABI is "m". You do not have to set this variable as the OpenEmbedded build system sets it for you.

The OpenEmbedded build system uses the ABI to construct directory names used when installing the Python headers and libraries in sysroot (e.g. `.../python3.3m/...`).

Recipes that inherit the `distutils` class during cross-builds also use this variable to locate the headers and libraries of the appropriate Python that the extension is targeting.

#### `PYTHON_PN`

When used by recipes that inherit the `distutils3`, `setuptools3`, `distutils`, or `setuptools` classes, specifies the major Python version being built. For Python 3.x, `PYTHON_PN` would be "python3". You do not have to set this variable as the OpenEmbedded build system automatically sets it for you.

The variable allows recipes to use common infrastructure such as the following:

```
DEPENDS += "${PYTHON_PN}-native"
```

In the previous example, the version of the dependency is `PYTHON_PN`.

## R

#### `RANLIB`

The minimal command and arguments to run `ranlib`.

#### `RCONFLICTS`

The list of packages that conflict with packages. Note that packages will not be installed if conflicting packages are not first removed.

Like all package-controlling variables, you must always use them in conjunction with a package name override. Here is an example:

```
RCONFLICTS_${PN} = "another_conflicting_package_name"
```

BitBake, which the OpenEmbedded build system uses, supports specifying versioned dependencies. Although the syntax varies depending on the packaging format, BitBake hides these differences from you. Here is the general syntax to specify versions with the `RCONFLICTS` variable:

```
RCONFLICTS_${PN} = "package (operator version)"
```

For `operator`, you can specify the following:

```
=
<
>
<=
>=
```

For example, the following sets up a dependency on version 1.2 or greater of the package `foo`:

```
RCONFLICTS_${PN} = "foo (>= 1.2)"
```

#### `RDEPENDS`

Lists runtime dependencies of a package. These dependencies are other packages that must be installed in order for the package to function correctly. As an example, the following assignment declares that the package `foo` needs the packages `bar` and `baz` to be installed:

```
RDEPENDS_foo = "bar baz"
```

The most common types of package runtime dependencies are automatically detected and added. Therefore, most recipes do not need to set `RDEPENDS`. For more information, see the "[Automatically Added Runtime Dependencies](#)" section in the Yocto Project Overview and Concepts Manual.

The practical effect of the above `RDEPENDS` assignment is that `bar` and `baz` will be declared as dependencies inside the package `foo` when it is written out by one of the `do_package_write_*` tasks. Exactly how this is done depends on which package format is used, which is determined by `PACKAGE_CLASSES`. When the corresponding package manager installs the package, it will know to also install the packages on which it depends.

To ensure that the packages `bar` and `baz` get built, the previous `RDEPENDS` assignment also causes a task dependency to be added. This dependency is from the recipe's `do_build` (not to be confused with `do_compile`) task to the `do_package_write_*` task of the recipes that build `bar` and `baz`.

The names of the packages you list within `RDEPENDS` must be the names of other packages - they cannot be recipe names. Although package names and recipe names

usually match, the important point here is that you are providing package names within the `RDEPENDS` variable. For an example of the default list of packages created from a recipe, see the [PACKAGES](#) variable.

Because the `RDEPENDS` variable applies to packages being built, you should always use the variable in a form with an attached package name (remember that a single recipe can build multiple packages). For example, suppose you are building a development package that depends on the `perl` package. In this case, you would use the following `RDEPENDS` statement:

```
RDEPENDS_${PN}-dev += "perl"
```

In the example, the development package depends on the `perl` package. Thus, the `RDEPENDS` variable has the `${PN}-dev` package name as part of the variable.

## Caution

`RDEPENDS_${PN}-dev` includes `${PN}` by default. This default is set in the BitBake configuration file (`meta/conf/bitbake.conf`). Be careful not to accidentally remove `${PN}` when modifying `RDEPENDS_${PN}-dev`. Use the `+="` operator rather than the `="` operator.

The package names you use with `RDEPENDS` must appear as they would in the `PACKAGES` variable. The `PKG` variable allows a different name to be used for the final package (e.g. the `debian` class uses this to rename packages), but this final package name cannot be used with `RDEPENDS`, which makes sense as `RDEPENDS` is meant to be independent of the package format used.

BitBake, which the OpenEmbedded build system uses, supports specifying versioned dependencies. Although the syntax varies depending on the packaging format, BitBake hides these differences from you. Here is the general syntax to specify versions with the `RDEPENDS` variable:

```
RDEPENDS_${PN} = "package (operator version)"
```

For *operator*, you can specify the following:

```
=
<
>
<=
>=
```

For *version*, provide the version number.

## Tip

You can use [EXTENDPKG](#) to provide a full package version specification.

For example, the following sets up a dependency on version 1.2 or greater of the package `foo`:

```
RDEPENDS_${PN} = "foo (>= 1.2)"
```

For information on build-time dependencies, see the [DEPENDS](#) variable. You can also see the ["Tasks"](#) and ["Dependencies"](#) sections in the BitBake User Manual for additional information on tasks and dependencies.

### REQUIRED\_DISTRO\_FEATURES¶

When inheriting the [distro\\_features\\_check](#) class, this variable identifies distribution features that must exist in the current configuration in order for the OpenEmbedded build system to build the recipe. In other words, if the `REQUIRED_DISTRO_FEATURES` variable lists a feature that does not appear in `DISTRO_FEATURES` within the current configuration, an error occurs and the build stops.

### RM\_WORK\_EXCLUDE¶

With `rm_work` enabled, this variable specifies a list of recipes whose work directories should not be removed. See the ["rm\\_work.bbclass"](#) section for more details.

### ROOT\_HOME¶

Defines the root home directory. By default, this directory is set as follows in the BitBake configuration file:

```
ROOT_HOME ??= "/home/root"
```

## Note

This default value is likely used because some embedded solutions prefer to have a read-only root filesystem and prefer to keep writeable data in one place.

You can override the default by setting the variable in any layer or in the `local.conf` file. Because the default is set using a "weak" assignment (i.e. `?? =`), you can use either of the following forms to define your override:

```
ROOT_HOME = "/root"
ROOT_HOME ?= "/root"
```

These override examples use `/root`, which is probably the most commonly used override.

|                                           |                                                                                                                                                                                                                                                                                                                                                         |
|-------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ROOTFS</code>                       | Indicates a filesystem image to include as the root filesystem.                                                                                                                                                                                                                                                                                         |
|                                           | The <code>ROOTFS</code> variable is an optional variable used with the <code>image-live</code> class.                                                                                                                                                                                                                                                   |
| <code>ROOTFS_POSTINSTALL_COMMAND</code>   | Specifies a list of functions to call after the OpenEmbedded build system has installed packages. You can specify functions separated by semicolons:                                                                                                                                                                                                    |
|                                           | <code>ROOTFS_POSTINSTALL_COMMAND += "function; ... "</code>                                                                                                                                                                                                                                                                                             |
|                                           | If you need to pass the root filesystem path to a command within a function, you can use <code>\${IMAGE_ROOTFS}</code> , which points to the directory that becomes the root filesystem image. See the <code>IMAGE_ROOTFS</code> variable for more information.                                                                                         |
| <code>ROOTFS_POSTPROCESS_COMMAND</code>   | Specifies a list of functions to call once the OpenEmbedded build system has created the root filesystem. You can specify functions separated by semicolons:                                                                                                                                                                                            |
|                                           | <code>ROOTFS_POSTPROCESS_COMMAND += "function; ... "</code>                                                                                                                                                                                                                                                                                             |
|                                           | If you need to pass the root filesystem path to a command within a function, you can use <code>\${IMAGE_ROOTFS}</code> , which points to the directory that becomes the root filesystem image. See the <code>IMAGE_ROOTFS</code> variable for more information.                                                                                         |
| <code>ROOTFS_POSTUNINSTALL_COMMAND</code> | Specifies a list of functions to call after the OpenEmbedded build system has removed unnecessary packages. When runtime package management is disabled in the image, several packages are removed including <code>base-passwd</code> , <code>shadow</code> , and <code>update-alternatives</code> . You can specify functions separated by semicolons: |
|                                           | <code>ROOTFS_POSTUNINSTALL_COMMAND += "function; ... "</code>                                                                                                                                                                                                                                                                                           |
|                                           | If you need to pass the root filesystem path to a command within a function, you can use <code>\${IMAGE_ROOTFS}</code> , which points to the directory that becomes the root filesystem image. See the <code>IMAGE_ROOTFS</code> variable for more information.                                                                                         |
| <code>ROOTFS_PREPROCESS_COMMAND</code>    | Specifies a list of functions to call before the OpenEmbedded build system has created the root filesystem. You can specify functions separated by semicolons:                                                                                                                                                                                          |
|                                           | <code>ROOTFS_PREPROCESS_COMMAND += "function; ... "</code>                                                                                                                                                                                                                                                                                              |
|                                           | If you need to pass the root filesystem path to a command within a function, you can use <code>\${IMAGE_ROOTFS}</code> , which points to the directory that becomes the root filesystem image. See the <code>IMAGE_ROOTFS</code> variable for more information.                                                                                         |
| <code>RPROVIDES</code>                    | A list of package name aliases that a package also provides. These aliases are useful for satisfying runtime dependencies of other packages both during the build and on the target (as specified by <code>RDEPENDS</code> ).                                                                                                                           |

## Note

A package's own name is implicitly already in its `RPROVIDES` list.

As with all package-controlling variables, you must always use the variable in conjunction with a package name override. Here is an example:

```
RPROVIDES_${PN} = "widget-abi-2"
```

|                          |                                                                                                                                                                                                                                                                                                                             |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>RRECOMMENDS</code> | A list of packages that extends the usability of a package being built. The package being built does not depend on this list of packages in order to successfully build, but rather uses them for extended usability. To specify runtime dependencies for packages, see the <code>RDEPENDS</code> variable.                 |
|                          | The package manager will automatically install the <code>RRECOMMENDS</code> list of packages when installing the built package. However, you can prevent listed packages from being installed by using the <code>BAD_RECOMMENDATIONS</code> , <code>NO_RECOMMENDATIONS</code> , and <code>PACKAGE_EXCLUDE</code> variables. |
|                          | Packages specified in <code>RRECOMMENDS</code> need not actually be produced. However, a recipe must exist that provides each package, either through the <code>PACKAGES</code> or <code>PACKAGES_DYNAMIC</code> variables or the <code>RPROVIDES</code> variable, or an error will                                         |

occur during the build. If such a recipe does exist and the package is not produced, the build continues without error.

Because the `RRECOMMENDS` variable applies to packages being built, you should always attach an override to the variable to specify the particular package whose usability is being extended. For example, suppose you are building a development package that is extended to support wireless functionality. In this case, you would use the following:

```
RRECOMMENDS_${PN}-dev += "wireless_package_name"
```

In the example, the package name (`${PN}-dev`) must appear as it would in the `PACKAGES` namespace before any renaming of the output package by classes such as `debian.bbclass`.

BitBake, which the OpenEmbedded build system uses, supports specifying versioned recommends. Although the syntax varies depending on the packaging format, BitBake hides these differences from you. Here is the general syntax to specify versions with the `RRECOMMENDS` variable:

```
RRECOMMENDS_${PN} = "package (operator version)"
```

For `operator`, you can specify the following:

```
=
<
>
<=
>=
```

For example, the following sets up a recommend on version 1.2 or greater of the package `foo`:

```
RRECOMMENDS_${PN} = "foo (>= 1.2)"
```

## RREPLACES¶

A list of packages replaced by a package. The package manager uses this variable to determine which package should be installed to replace other package(s) during an upgrade. In order to also have the other package(s) removed at the same time, you must add the name of the other package to the `RCONFLICTS` variable.

As with all package-controlling variables, you must use this variable in conjunction with a package name override. Here is an example:

```
RREPLACES_${PN} = "other_package_being_replaced"
```

BitBake, which the OpenEmbedded build system uses, supports specifying versioned replacements. Although the syntax varies depending on the packaging format, BitBake hides these differences from you. Here is the general syntax to specify versions with the `RREPLACES` variable:

```
RREPLACES_${PN} = "package (operator version)"
```

For `operator`, you can specify the following:

```
=
<
>
<=
>=
```

For example, the following sets up a replacement using version 1.2 or greater of the package `foo`:

```
RREPLACES_${PN} = "foo (>= 1.2)"
```

## RSUGGESTS¶

A list of additional packages that you can suggest for installation by the package manager at the time a package is installed. Not all package managers support this functionality.

As with all package-controlling variables, you must always use this variable in conjunction with a package name override. Here is an example:

```
RSUGGESTS_${PN} = "useful_package another_package"
```

## S¶

### S¶

The location in the [Build Directory](#) where unpacked recipe source code resides. By default, this directory is `${WORKDIR}/${BPN}-${PV}`, where `${BPN}` is the base recipe name and `${PV}` is the recipe version. If the source tarball extracts the code to a directory named anything other than `${BPN}-${PV}`, or if the source code is fetched from an SCM such as Git or Subversion, then you must set `S` in the recipe so that the OpenEmbedded build system knows where to find the unpacked source.

As an example, assume a [Source Directory](#) top-level folder named `poky` and a default Build Directory at `poky/build`. In this case, the work directory the build



system uses to keep the unpacked recipe for `db` is the following:

```
poky/build/tmp/work/qemux86-poky-linux/db/5.1.19-r3/db-5.1.19
```

The unpacked source code resides in the `db-5.1.19` folder.

This next example assumes a Git repository. By default, Git repositories are cloned to `${WORKDIR}/git` during `do_fetch`. Since this path is different from the default value of `S`, you must set it specifically so the source can be located:

```
SRC_URI = "git://path/to/repo.git"
S = "${WORKDIR}/git"
```

#### `SANITY_REQUIRED_UTILITIES`

Specifies a list of command-line utilities that should be checked for during the initial sanity checking process when running BitBake. If any of the utilities are not installed on the build host, then BitBake immediately exits with an error.

#### `SANITY_TESTED_DISTROS`

A list of the host distribution identifiers that the build system has been tested against. Identifiers consist of the host distributor ID followed by the release, as reported by the `lsb_release` tool or as read from `/etc/lsb-release`. Separate the list items with explicit newline characters (`\n`). If `SANITY_TESTED_DISTROS` is not empty and the current value of `NATIVE_LSB_STRING` does not appear in the list, then the build system reports a warning that indicates the current host distribution has not been tested as a build host.

#### `SDK_ARCH`

The target architecture for the SDK. Typically, you do not directly set this variable. Instead, use `SDKMACHINE`.

#### `SDK_DEPLOY`

The directory set up and used by the `populate_sdk_base` class to which the SDK is deployed. The `populate_sdk_base` class defines `SDK_DEPLOY` as follows:

```
SDK_DEPLOY = "${TMPDIR}/deploy/sdk"
```

#### `SDK_DIR`

The parent directory used by the OpenEmbedded build system when creating SDK output. The `populate_sdk_base` class defines the variable as follows:

```
SDK_DIR = "${WORKDIR}/sdk"
```

### Note

The `SDK_DIR` directory is a temporary directory as it is part of `WORKDIR`. The final output directory is `SDK_DEPLOY`.

#### `SDK_EXT_TYPE`

Controls whether or not shared state artifacts are copied into the extensible SDK. The default value of "full" copies all of the required shared state artifacts into the extensible SDK. The value "minimal" leaves these artifacts out of the SDK.

### Note

If you set the variable to "minimal", you need to ensure `STATE_MIRRORS` is set in the SDK's configuration to enable the artifacts to be fetched as needed.

#### `SDK_HOST_MANIFEST`

The manifest file for the host part of the SDK. This file lists all the installed packages that make up the host part of the SDK. The file contains package information on a line-per-package basis as follows:

```
packagename packagearch version
```

The `populate_sdk_base` class defines the manifest file as follows:

```
SDK_HOST_MANIFEST = "${SDK_DEPLOY}/${TOOLCHAIN_OUTPUTNAME}.host.manifest"
```

The location is derived using the `SDK_DEPLOY` and `TOOLCHAIN_OUTPUTNAME` variables.

#### `SDK_INCLUDE_PKGDATA`

When set to "1", specifies to include the packagedata for all recipes in the "world" target in the extensible SDK. Including this data allows the `devtool search` command to find these recipes in search results, as well as allows the `devtool add` command to map dependencies more effectively.

### Note

Enabling the `SDK_INCLUDE_PKGDATA` variable significantly increases build time because all of world needs

to be built. Enabling the variable also slightly increases the size of the extensible SDK.

#### SDK\_INCLUDE\_TOOLCHAIN¶

When set to "1", specifies to include the toolchain in the extensible SDK. Including the toolchain is useful particularly when `SDK_EXT_TYPE` is set to "minimal" to keep the SDK reasonably small but you still want to provide a usable toolchain. For example, suppose you want to use the toolchain from an IDE or from other tools and you do not want to perform additional steps to install the toolchain.

The `SDK_INCLUDE_TOOLCHAIN` variable defaults to "0" if `SDK_EXT_TYPE` is set to "minimal", and defaults to "1" if `SDK_EXT_TYPE` is set to "full".

#### SDK\_INHERIT\_BLACKLIST¶

A list of classes to remove from the `INHERIT` value globally within the extensible SDK configuration. The `populate-sdk-ext` class sets the default value:

```
SDK_INHERIT_BLACKLIST ?= "buildhistory icecc"
```

Some classes are not generally applicable within the extensible SDK context. You can use this variable to disable those classes.

For additional information on how to customize the extensible SDK's configuration, see the "[Configuring the Extensible SDK](#)" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

#### SDK\_LOCAL\_CONF\_BLACKLIST¶

A list of variables not allowed through from the OpenEmbedded build system configuration into the extensible SDK configuration. Usually, these are variables that are specific to the machine on which the build system is running and thus would be potentially problematic within the extensible SDK.

By default, `SDK_LOCAL_CONF_BLACKLIST` is set in the `populate-sdk-ext` class and excludes the following variables:

```
CONF_VERSION
BB_NUMBER_THREADS
BB_NUMBER_PARSE_THREADS
PARALLEL_MAKE
PRSERV_HOST
SSTATE_MIRRORS
DL_DIR
SSTATE_DIR
TMPDIR
BB_SERVER_TIMEOUT
```

For additional information on how to customize the extensible SDK's configuration, see the "[Configuring the Extensible SDK](#)" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

#### SDK\_LOCAL\_CONF\_WHITELIST¶

A list of variables allowed through from the OpenEmbedded build system configuration into the extensible SDK configuration. By default, the list of variables is empty and is set in the `populate-sdk-ext` class.

This list overrides the variables specified using the `SDK_LOCAL_CONF_BLACKLIST` variable as well as any variables identified by automatic blacklisting due to the "/" character being found at the start of the value, which is usually indicative of being a path and thus might not be valid on the system where the SDK is installed.

For additional information on how to customize the extensible SDK's configuration, see the "[Configuring the Extensible SDK](#)" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

#### SDK\_NAME¶

The base name for SDK output files. The name is derived from the `DISTRO`, `TCLIBC`, `SDK_ARCH`, `IMAGE_BASENAME`, and `TUNE_PKGARCH` variables:

```
SDK_NAME = "${DISTRO}-${TCLIBC}-${SDK_ARCH}-${IMAGE_BASENAME}-${TUNE_PKGARCH}
```



#### SDK\_OS¶

Specifies the operating system for which the SDK will be built. The default value is the value of `BUILD_OS`.

#### SDK\_OUTPUT¶

The location used by the OpenEmbedded build system when creating SDK output. The `populate_sdk_base` class defines the variable as follows:

```
SDK_DIR = "${WORKDIR}/sdk"
SDK_OUTPUT = "${SDK_DIR}/image"
SDK_DEPLOY = "${DEPLOY_DIR}/sdk"
```

### Note

The `SDK_OUTPUT` directory is a temporary directory as it is part of `WORKDIR` by way of `SDK_DIR`. The final output directory is `SDK_DEPLOY`.

|                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SDK_PACKAGE_ARCHS¶       | <p>Specifies a list of architectures compatible with the SDK machine. This variable is set automatically and should not normally be hand-edited. Entries are separated using spaces and listed in order of priority. The default value for <code>SDK_PACKAGE_ARCHS</code> is "all any noarch \${SDK_ARCH}-\${SDKPKGSUFFIX}".</p>                                                                                                                                                                                                                                                                                                                                                                   |
| SDK_POSTPROCESS_COMMAND¶ | <p>Specifies a list of functions to call once the OpenEmbedded build system creates the SDK. You can specify functions separated by semicolons:</p> <pre>SDK_POSTPROCESS_COMMAND += "function; ... "</pre> <p>If you need to pass an SDK path to a command within a function, you can use <code>\${SDK_DIR}</code>, which points to the parent directory used by the OpenEmbedded build system when creating SDK output. See the <a href="#">SDK_DIR</a> variable for more information.</p>                                                                                                                                                                                                        |
| SDK_PREFIX¶              | <p>The toolchain binary prefix used for <code>nativesdk</code> recipes. The OpenEmbedded build system uses the <code>SDK_PREFIX</code> value to set the <a href="#">TARGET_PREFIX</a> when building <code>nativesdk</code> recipes. The default value is "\${SDK_SYS}-".</p>                                                                                                                                                                                                                                                                                                                                                                                                                       |
| SDK_RECRDEP_TASKS¶       | <p>A list of shared state tasks added to the extensible SDK. By default, the following tasks are added:</p> <pre>do_populate_lic do_package_qa do_populate_sysroot do_deploy</pre> <p>Despite the default value of "" for the <code>SDK_RECRDEP_TASKS</code> variable, the above four tasks are always added to the SDK. To specify tasks beyond these four, you need to use the <code>SDK_RECRDEP_TASKS</code> variable (e.g. you are defining additional tasks that are needed in order to build <a href="#">SDK_TARGETS</a>).</p>                                                                                                                                                               |
| SDK_SYS¶                 | <p>Specifies the system, including the architecture and the operating system, for which the SDK will be built.</p> <p>The OpenEmbedded build system automatically sets this variable based on <a href="#">SDK_ARCH</a>, <a href="#">SDK_VENDOR</a>, and <a href="#">SDK_OS</a>. You do not need to set the <code>SDK_SYS</code> variable yourself.</p>                                                                                                                                                                                                                                                                                                                                             |
| SDK_TARGET_MANIFEST¶     | <p>The manifest file for the target part of the SDK. This file lists all the installed packages that make up the target part of the SDK. The file contains package information on a line-per-package basis as follows:</p> <pre>packagename packagearch version</pre> <p>The <code>populate_sdk_base</code> class defines the manifest file as follows:</p> <pre>SDK_TARGET_MANIFEST = "\${SDK_DEPLOY}/\${TOOLCHAIN_OUTPUTNAME}.target.manifest</pre> <div></div> <p>The location is derived using the <a href="#">SDK_DEPLOY</a> and <a href="#">TOOLCHAIN_OUTPUTNAME</a> variables.</p>                                                                                                          |
| SDK_TARGETS¶             | <p>A list of targets to install from shared state as part of the standard or extensible SDK installation. The default value is "\${PN}" (i.e. the image from which the SDK is built).</p> <p>The <code>SDK_TARGETS</code> variable is an internal variable and typically would not be changed.</p>                                                                                                                                                                                                                                                                                                                                                                                                 |
| SDK_TITLE¶               | <p>The title to be printed when running the SDK installer. By default, this title is based on the <a href="#">DISTRO_NAME</a> or <a href="#">DISTRO</a> variable and is set in the <code>populate_sdk_base</code> class as follows:</p> <pre>SDK_TITLE ??= "\${@d.getVar('DISTRO_NAME') or d.getVar('DISTRO')}" SDK"</pre> <p>For the default distribution "poky", <code>SDK_TITLE</code> is set to "Poky (Yocto Project Reference Distro)".</p> <p>For information on how to change this default title, see the "<a href="#">Changing the Extensible SDK Installer Title</a>" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.</p> |
| SDK_UPDATE_URL¶          | <p>An optional URL for an update server for the extensible SDK. If set, the value is used as the default update server when running <code>devtool sdk-update</code> within the extensible SDK.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| SDK_VENDOR¶              | <p>Specifies the name of the SDK vendor.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| SDK_VERSION¶             | <p>Specifies the version of the SDK. The distribution configuration file (e.g. <code>/meta-poky/conf/distro/poky.conf</code>) defines the <code>SDK_VERSION</code> as follows:</p> <pre>SDK_VERSION = "\${@d.getVar('DISTRO_VERSION')}.replace(' snapshot-\${DATE}', 'sn</pre> <div></div> <p>For additional information, see the <a href="#">DISTRO_VERSION</a> and <a href="#">DATE</a> variables.</p>                                                                                                                                                                                                                                                                                           |
| SDKEXTPATH¶              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

The default installation directory for the Extensible SDK. By default, this directory is based on the `DISTRO` variable and is set in the `populate_sdk_base` class as follows:

```
SDKEXTPATH ??= "${@d.getVar('DISTRO')}_sdk"
```

For the default distribution "poky", the `SDKEXTPATH` is set to "poky\_sdk".

For information on how to change this default directory, see the "[Changing the Default SDK Installation Directory](#)" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

#### SDKIMAGE\_FEATURES¶

Equivalent to `IMAGE_FEATURES`. However, this variable applies to the SDK generated from an image using the following command:

```
$ bitbake -c populate_sdk imagename
```

#### SDKMACHINE¶

The machine for which the SDK is built. In other words, the SDK is built such that it runs on the target you specify with the `SDKMACHINE` value. The value points to a corresponding `.conf` file under `conf/machine-sdk/`.

You can use "i686" and "x86\_64" as possible values for this variable. The variable defaults to "i686" and is set in the `local.conf` file in the Build Directory.

```
SDKMACHINE ?= "i686"
```

### Note

You cannot set the `SDKMACHINE` variable in your distribution configuration file. If you do, the configuration will not take affect.

#### SDKPATH¶

Defines the path offered to the user for installation of the SDK that is generated by the OpenEmbedded build system. The path appears as the default location for installing the SDK when you run the SDK's installation script. You can override the offered path when you run the script.

#### SDKTARGETSYSROOT¶

The full path to the sysroot used for cross-compilation within an SDK as it will be when installed into the default `SDKPATH`.

#### SECTION¶

The section in which packages should be categorized. Package management utilities can make use of this variable.

#### SELECTED\_OPTIMIZATION¶

Specifies the optimization flags passed to the C compiler when building for the target. The flags are passed through the default value of the `TARGET_CFLAGS` variable.

The `SELECTED_OPTIMIZATION` variable takes the value of `FULL_OPTIMIZATION` unless `DEBUG_BUILD` = "1". If that is the case, the value of `DEBUG_OPTIMIZATION` is used.

#### SERIAL\_CONSOLE¶

Defines a serial console (TTY) to enable using `getty`. Provide a value that specifies the baud rate followed by the TTY device name separated by a space. You cannot specify more than one TTY device:

```
SERIAL_CONSOLE = "115200 ttyS0"
```

### Note

The `SERIAL_CONSOLE` variable is deprecated. Please use the `SERIAL_CONSOLES` variable.

#### SERIAL\_CONSOLES¶

Defines a serial console (TTY) to enable using `getty`. Provide a value that specifies the baud rate followed by the TTY device name separated by a semicolon. Use spaces to separate multiple devices:

```
SERIAL_CONSOLES = "115200;ttyS0 115200;ttyS1"
```

#### SERIAL\_CONSOLES\_CHECK¶

Specifies serial consoles, which must be listed in `SERIAL_CONSOLES`, to check against `/proc/console` before enabling them using `getty`. This variable allows aliasing in the format: `<device>:<alias>`. If a device was listed as "sclp\_line0" in `/dev/` and "ttyS0" was listed in `/proc/console`, you would do the following:

```
SERIAL_CONSOLES_CHECK = "sclp_line0:ttyS0"
```

This variable is currently only supported with SysVinit (i.e. not with systemd).

#### SIGGEN\_EXCLUDE\_SAFE\_RECIPE\_DEPS¶

A list of recipe dependencies that should not be used to determine signatures of tasks from one recipe when they depend on tasks from another recipe. For example:

```
SIGGEN_EXCLUDE_SAFE_RECIPE_DEPS += "intone->mplayer2"
```

In the previous example, `intone` depends on `mplayer2`.

You can use the special token `"*"` on the left-hand side of the dependency to match all recipes except the one on the right-hand side. Here is an example:

```
SIGGEN_EXCLUDE_SAFE_RECIPE_DEPS += "*->quilt-native"
```

In the previous example, all recipes except `quilt-native` ignore task signatures from the `quilt-native` recipe when determining their task signatures.

Use of this variable is one mechanism to remove dependencies that affect task signatures and thus force rebuilds when a recipe changes.

### Caution

If you add an inappropriate dependency for a recipe relationship, the software might break during runtime if the interface of the second recipe was changed after the first recipe had been built.

`SIGGEN_EXCLUDE_RECIPES_ABISAFE`

A list of recipes that are completely stable and will never change. The ABI for the recipes in the list are presented by output from the tasks run to build the recipe. Use of this variable is one way to remove dependencies from one recipe on another that affect task signatures and thus force rebuilds when the recipe changes.

### Caution

If you add an inappropriate variable to this list, the software might break at runtime if the interface of the recipe was changed after the other had been built.

`SITEINFO_BITS`

Specifies the number of bits for the target system CPU. The value should be either "32" or "64".

`SITEINFO_ENDIANNESS`

Specifies the endian byte order of the target system. The value should be either "le" for little-endian or "be" for big-endian.

`SKIP_FILEDEPS`

Enables removal of all files from the "Provides" section of an RPM package. Removal of these files is required for packages containing prebuilt binaries and libraries such as `libstdc++` and `glibc`.

To enable file removal, set the variable to "1" in your `conf/local.conf` configuration file in your: [Build Directory](#).

```
SKIP_FILEDEPS = "1"
```

`SOC_FAMILY`

Groups together machines based upon the same family of SOC (System On Chip). You typically set this variable in a common `.inc` file that you include in the configuration files of all the machines.

### Note

You must include `conf/machine/include/soc-family.inc` for this variable to appear in [MACHINEOVERRIDES](#).

`SOLIBS`

Defines the suffix for shared libraries used on the target platform. By default, this suffix is `".so.*"` for all Linux-based systems and is defined in the `meta/conf/bitbake.conf` configuration file.

You will see this variable referenced in the default values of `FILES_${PN}`.

`SOLIBSDEV`

Defines the suffix for the development symbolic link (symlink) for shared libraries on the target platform. By default, this suffix is `".so"` for Linux-based systems and is defined in the `meta/conf/bitbake.conf` configuration file.

You will see this variable referenced in the default values of `FILES_${PN}-dev`.

`SOURCE_MIRROR_FETCH`

When you are fetching files to create a mirror of sources (i.e. creating a source mirror), setting `SOURCE_MIRROR_FETCH` to "1" in your `local.conf` configuration file ensures the source for all recipes are fetched regardless of whether or not a recipe is compatible with the configuration. A recipe is considered incompatible with the currently configured machine when either or both the

`COMPATIBLE_MACHINE` variable and `COMPATIBLE_HOST` variables specify compatibility with a machine other than that of the current machine or host.

## Warning

Do not set the `SOURCE_MIRROR_FETCH` variable unless you are creating a source mirror. In other words, do not set the variable during a normal build.

### `SOURCE_MIRROR_URL`

Defines your own `PREMIRRORS` from which to first fetch source before attempting to fetch from the upstream specified in `SRC_URI`.

To use this variable, you must globally inherit the `own-mirrors` class and then provide the URL to your mirrors. Here is the general syntax:

```
INHERIT += "own-mirrors"
SOURCE_MIRROR_URL = "http://example.com/my_source_mirror"
```

## Note

You can specify only a single URL in `SOURCE_MIRROR_URL`.

### `SPDXLICENSEMAP`

Maps commonly used license names to their SPDX counterparts found in `meta/files/common-licenses/`. For the default `SPDXLICENSEMAP` mappings, see the `meta/conf/licenses.conf` file.

For additional information, see the `LICENSE` variable.

### `SPECIAL_PKGSUFFIX`

A list of prefixes for `PN` used by the OpenEmbedded build system to create variants of recipes or packages. The list specifies the prefixes to strip off during certain circumstances such as the generation of the `BPN` variable.

### `SPL_BINARY`

The file type for the Secondary Program Loader (SPL). Some devices use an SPL from which to boot (e.g. the BeagleBone development board). For such cases, you can declare the file type of the SPL binary in the `u-boot.inc` include file, which is used in the U-Boot recipe.

The SPL file type is set to "null" by default in the `u-boot.inc` file as follows:

```
Some versions of u-boot build an SPL (Second Program Loader) image that
should be packaged along with the u-boot binary as well as placed in the
deploy directory. For those versions they can set the following variable
to allow packaging the SPL.
SPL_BINARY ?= ""
SPL_BINARYNAME ?= "${@os.path.basename(d.getVar("SPL_BINARY"))}"
SPL_IMAGE ?= "${SPL_BINARYNAME}-${MACHINE}-${PV}-${PR}"
SPL_SYMLINK ?= "${SPL_BINARYNAME}-${MACHINE}"
```

The `SPL_BINARY` variable helps form various `SPL_*` variables used by the OpenEmbedded build system.

See the BeagleBone machine configuration example in the "[Creating a new BSP Layer Using the bitbake-layers Script](#)" section in the Yocto Project Board Support Package Developer's Guide for additional information.

### `SRC_URI`

The list of source files - local or remote. This variable tells the OpenEmbedded build system which bits to pull in for the build and how to pull them in. For example, if the recipe or append file only needs to fetch a tarball from the Internet, the recipe or append file uses a single `SRC_URI` entry. On the other hand, if the recipe or append file needs to fetch a tarball, apply two patches, and include a custom file, the recipe or append file would include four instances of the variable.

The following list explains the available URI protocols. URI protocols are highly dependent on particular BitBake Fetcher submodules. Depending on the fetcher BitBake uses, various URL parameters are employed. For specifics on the supported Fetchers, see the "[Fetchers](#)" section in the BitBake User Manual.

- **`file://`** - Fetches files, which are usually files shipped with the [Metadata](#), from the local machine (e.g. [patch](#) files). The path is relative to the `FILES_PATH` variable. Thus, the build system searches, in order, from the following directories, which are assumed to be a subdirectories of the directory in which the recipe file (`.bb`) or append file (`.bbappend`) resides:
  - `_${BPN}` - The base recipe name without any special suffix or version numbers.
  - `_${BP}` - `_${BPN} - ${PV}`. The base recipe name and version but without any special package name suffix.
  - `files` - Files within a directory, which is named `files` and is also alongside the recipe or append file.

## Note

If you want the build system to pick up files specified through a `SRC_URI` statement from your append file, you need to be sure to extend the `FILESPATH` variable by also using the `FILESEXTRAPATHS` variable from within your append file.

- **bzr://** - Fetches files from a Bazaar revision control repository.
- **git://** - Fetches files from a Git revision control repository.
- **osc://** - Fetches files from an OSC (OpenSUSE Build service) revision control repository.
- **repo://** - Fetches files from a repo (Git) repository.
- **ccrc://** - Fetches files from a ClearCase repository.
- **http://** - Fetches files from the Internet using `http`.
- **https://** - Fetches files from the Internet using `https`.
- **ftp://** - Fetches files from the Internet using `ftp`.
- **cvs://** - Fetches files from a CVS revision control repository.
- **hg://** - Fetches files from a Mercurial (`hg`) revision control repository.
- **p4://** - Fetches files from a Perforce (`p4`) revision control repository.
- **ssh://** - Fetches files from a secure shell.
- **svn://** - Fetches files from a Subversion (`svn`) revision control repository.
- **npm://** - Fetches JavaScript modules from a registry.

Standard and recipe-specific options for `SRC_URI` exist. Here are standard options:

- **apply** - Whether to apply the patch or not. The default action is to apply the patch.
- **striplevel** - Which striplevel to use when applying the patch. The default level is 1.
- **patchdir** - Specifies the directory in which the patch should be applied. The default is `${S}`.

Here are options specific to recipes building code from a revision control system:

- **mindate** - Apply the patch only if `SRCDATE` is equal to or greater than `mindate`.
- **maxdate** - Apply the patch only if `SRCDATE` is not later than `maxdate`.
- **minrev** - Apply the patch only if `SRCREV` is equal to or greater than `minrev`.
- **maxrev** - Apply the patch only if `SRCREV` is not later than `maxrev`.
- **rev** - Apply the patch only if `SRCREV` is equal to `rev`.
- **notrev** - Apply the patch only if `SRCREV` is not equal to `rev`.

Here are some additional options worth mentioning:

- **unpack** - Controls whether or not to unpack the file if it is an archive. The default action is to unpack the file.
- **destsuffix** - Places the file (or extracts its contents) into the specified subdirectory of `WORKDIR` when the Git fetcher is used.
- **subdir** - Places the file (or extracts its contents) into the specified subdirectory of `WORKDIR` when the local (`file://`) fetcher is used.
- **localdir** - Places the file (or extracts its contents) into the specified subdirectory of `WORKDIR` when the CVS fetcher is used.
- **subpath** - Limits the checkout to a specific subpath of the tree when using the Git fetcher is used.
- **name** - Specifies a name to be used for association with `SRC_URI` checksums when you have more than one file specified in `SRC_URI`.
- **downloadfilename** - Specifies the filename used when storing the downloaded file.

`SRC_URI_OVERRIDES_PACKAGE_ARCH`

By default, the OpenEmbedded build system automatically detects whether `SRC_URI` contains files that are machine-specific. If so, the build system automatically changes `PACKAGE_ARCH`. Setting this variable to "0" disables this behavior.

`SRCDATE`

**SRCPV**

The date of the source code used to build the package. This variable applies only if the source was fetched from a Source Code Manager (SCM).

Returns the version string of the current package. This string is used to help define the value of `PV`.

The `SRCPV` variable is defined in the `meta/conf/bitbake.conf` configuration file in the [Source Directory](#) as follows:

```
SRCPV = "${@bb.fetch2.get_srcrev(d)}"
```

Recipes that need to define `PV` do so with the help of the `SRCPV`. For example, the `ofono` recipe (`ofono_git.bb`) located in `meta/recipes-connectivity` in the Source Directory defines `PV` as follows:

```
PV = "0.12-git${SRCPV}"
```

**SRCREV**

The revision of the source code used to build the package. This variable applies to Subversion, Git, Mercurial, and Bazaar only. Note that if you want to build a fixed revision and you want to avoid performing a query on the remote repository every time BitBake parses your recipe, you should specify a `SRCREV` that is a full revision identifier and not just a tag.

**Note**

For information on limitations when inheriting the latest revision of software using `SRCREV`, see the [AUTOREV](#) variable description and the "[Automatically Incrementing a Binary Package Revision Number](#)" section, which is in the Yocto Project Development Tasks Manual.

**SSTATE\_DIR**

The directory for the shared state cache.

**SSTATE\_MIRROR\_ALLOW\_NETWORK**

If set to "1", allows fetches from mirrors that are specified in [SSTATE\\_MIRRORS](#) to work even when fetching from the network is disabled by setting `BB_NO_NETWORK` to "1". Using the `SSTATE_MIRROR_ALLOW_NETWORK` variable is useful if you have set `SSTATE_MIRRORS` to point to an internal server for your shared state cache, but you want to disable any other fetching from the network.

**SSTATE\_MIRRORS**

Configures the OpenEmbedded build system to search other mirror locations for prebuilt cache data objects before building out the data. This variable works like fetcher [MIRRORS](#) and [PREMIRRORS](#) and points to the cache locations to check for the shared state (sstate) objects.

You can specify a filesystem directory or a remote URL such as HTTP or FTP. The locations you specify need to contain the shared state cache (sstate-cache) results from previous builds. The sstate-cache you point to can also be from builds on other machines.

When pointing to sstate build artifacts on another machine that uses a different GCC version for native builds, you must configure `SSTATE_MIRROR` with a regular expression that maps local search paths to server paths. The paths need to take into account `NATIVELSBSTRING` set by the `uninative` class. For example, the following maps the local search path `universal-4.9` to the server-provided path `server_url_sstate_path`:

```
SSTATE_MIRRORS ?= file://universal-4.9/(.*) http://server_url_sstate_path/u
```



If a mirror uses the same structure as [SSTATE\\_DIR](#), you need to add "PATH" at the end as shown in the examples below. The build system substitutes the correct path within the directory structure.

```
SSTATE_MIRRORS ?= "\
file://.* http://someserver.tld/share/sstate/PATH;downloadfilename=PATH \n
file://.* file:///some-local-dir/sstate/PATH"
```

**SSTATE\_SCAN\_FILES**

Controls the list of files the OpenEmbedded build system scans for hardcoded installation paths. The variable uses a space-separated list of filenames (not paths) with standard wildcard characters allowed.

During a build, the OpenEmbedded build system creates a shared state (sstate) object during the first stage of preparing the sysroots. That object is scanned for hardcoded paths for original installation locations. The list of files that are scanned for paths is controlled by the `SSTATE_SCAN_FILES` variable. Typically, recipes add files they want to be scanned to the value of `SSTATE_SCAN_FILES` rather than the variable being comprehensively set. The [sstate](#) class specifies the default list of files.

For details on the process, see the [staging](#) class.

**STAGING\_BASE\_LIBDIR\_NATIVE**



|                                                                                                                                                                                                                                                                                                                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                                                                                                                                                                                                                                                                                | Specifies the path to the <code>/lib</code> subdirectory of the sysroot directory for the build host.                                                                                                                                                                                                                                                                                                                                                                        |
| <code>STAGING_BASELIBDIR</code>                                                                                                                                                                                                                                                                                | Specifies the path to the <code>/lib</code> subdirectory of the sysroot directory for the target for which the current recipe is being built ( <code>STAGING_DIR_HOST</code> ).                                                                                                                                                                                                                                                                                              |
| <code>STAGING_BINDIR</code>                                                                                                                                                                                                                                                                                    | Specifies the path to the <code>/usr/bin</code> subdirectory of the sysroot directory for the target for which the current recipe is being built ( <code>STAGING_DIR_HOST</code> ).                                                                                                                                                                                                                                                                                          |
| <code>STAGING_BINDIR_CROSS</code>                                                                                                                                                                                                                                                                              | Specifies the path to the directory containing binary configuration scripts. These scripts provide configuration information for other software that wants to make use of libraries or include files provided by the software associated with the script.                                                                                                                                                                                                                    |
| <p><b>Note</b></p> <p>This style of build configuration has been largely replaced by <code>pkg-config</code>. Consequently, if <code>pkg-config</code> is supported by the library to which you are linking, it is recommended you use <code>pkg-config</code> instead of a provided configuration script.</p> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <code>STAGING_BINDIR_NATIVE</code>                                                                                                                                                                                                                                                                             | Specifies the path to the <code>/usr/bin</code> subdirectory of the sysroot directory for the build host.                                                                                                                                                                                                                                                                                                                                                                    |
| <code>STAGING_DATADIR</code>                                                                                                                                                                                                                                                                                   | Specifies the path to the <code>/usr/share</code> subdirectory of the sysroot directory for the target for which the current recipe is being built ( <code>STAGING_DIR_HOST</code> ).                                                                                                                                                                                                                                                                                        |
| <code>STAGING_DATADIR_NATIVE</code>                                                                                                                                                                                                                                                                            | Specifies the path to the <code>/usr/share</code> subdirectory of the sysroot directory for the build host.                                                                                                                                                                                                                                                                                                                                                                  |
| <code>STAGING_DIR</code>                                                                                                                                                                                                                                                                                       | <p>Helps construct the <code>recipe-sysroots</code> directory, which is used during packaging.</p> <p>For information on how staging for recipe-specific sysroots occurs, see the <code>do_populate_sysroot</code> task, the "Sharing Files Between Recipes" section in the Yocto Project Development Tasks Manual, the "Configuration, Compilation, and Staging" section in the Yocto Project Overview and Concepts Manual, and the <code>SYSROOT_DIRS</code> variable.</p> |

### Note

Recipes should never write files directly under the `STAGING_DIR` directory because the OpenEmbedded build system manages the directory automatically. Instead, files should be installed to `${D}` within your recipe's `do_install` task and then the OpenEmbedded build system will stage a subset of those files into the sysroot.

|                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>STAGING_DIR_HOST</code> | <p>Specifies the path to the sysroot directory for the system on which the component is built to run (the system that hosts the component). For most recipes, this sysroot is the one in which that recipe's <code>do_populate_sysroot</code> task copies files. Exceptions include <code>-native</code> recipes, where the <code>do_populate_sysroot</code> task instead uses <code>STAGING_DIR_NATIVE</code>. Depending on the type of recipe and the build target, <code>STAGING_DIR_HOST</code> can have the following values:</p> <ul style="list-style-type: none"> <li>For recipes building for the target machine, the value is <code>"\${STAGING_DIR}/\${MACHINE}"</code>.</li> <li>For native recipes building for the build host, the value is empty given the assumption that when building for the build host, the build host's own directories should be used.</li> </ul> |
|-------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Note

`-native` recipes are not installed into host paths like such as `/usr`. Rather, these recipes are installed into `STAGING_DIR_NATIVE`. When compiling `-native` recipes, standard build environment variables such as `CPPFLAGS` and `CFLAGS` are set up so that both host paths and `STAGING_DIR_NATIVE` are searched for libraries and headers using, for example, GCC's `-isystem` option.

Thus, the emphasis is that the `STAGING_DIR*` variables should be viewed as input variables by tasks such as `do_configure`, `do_compile`, and `do_install`. Having the real system root correspond to `STAGING_DIR_HOST` makes conceptual sense for `-native` recipes, as they make use of host headers and libraries.

|                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|---------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>STAGING_DIR_NATIVE</code>       | Specifies the path to the sysroot directory used when building components that run on the build host itself.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <code>STAGING_DIR_TARGET</code>       | <p>Specifies the path to the sysroot used for the system for which the component generates code. For components that do not generate code, which is the majority, <code>STAGING_DIR_TARGET</code> is set to match <code>STAGING_DIR_HOST</code>.</p> <p>Some recipes build binaries that can run on the target system but those binaries in turn generate code for another different system (e.g. cross-canadian recipes). Using terminology from GNU, the primary system is referred to as the "HOST" and the secondary, or different, system is referred to as the "TARGET". Thus, the binaries run on the "HOST" system and generate binaries for the "TARGET" system. The <code>STAGING_DIR_HOST</code> variable points to the sysroot used for the "HOST" system, while <code>STAGING_DIR_TARGET</code> points to the sysroot used for the "TARGET" system.</p> |
| <code>STAGING_ETC_DIR_NATIVE</code>   | Specifies the path to the <code>/etc</code> subdirectory of the sysroot directory for the build host.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <code>STAGING_EXECPREFIXDIR</code>    | Specifies the path to the <code>/usr</code> subdirectory of the sysroot directory for the target for which the current recipe is being built ( <code>STAGING_DIR_HOST</code> ).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <code>STAGING_INCDIR</code>           | Specifies the path to the <code>/usr/include</code> subdirectory of the sysroot directory for the target for which the current recipe being built ( <code>STAGING_DIR_HOST</code> ).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <code>STAGING_INCDIR_NATIVE</code>    | Specifies the path to the <code>/usr/include</code> subdirectory of the sysroot directory for the build host.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <code>STAGING_KERNEL_BUILDDIR</code>  | Points to the directory containing the kernel build artifacts. Recipes building software that needs to access kernel build artifacts (e.g. <code>systemtap-uprobes</code> ) can look in the directory specified with the <code>STAGING_KERNEL_BUILDDIR</code> variable to find these artifacts after the kernel has been built.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <code>STAGING_KERNEL_DIR</code>       | The directory with kernel headers that are required to build out-of-tree modules.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <code>STAGING_LIBDIR</code>           | Specifies the path to the <code>/usr/lib</code> subdirectory of the sysroot directory for the target for which the current recipe is being built ( <code>STAGING_DIR_HOST</code> ).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <code>STAGING_LIBDIR_NATIVE</code>    | Specifies the path to the <code>/usr/lib</code> subdirectory of the sysroot directory for the build host.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <code>STAMP</code>                    | <p>Specifies the base path used to create recipe stamp files. The path to an actual stamp file is constructed by evaluating this string and then appending additional information. Currently, the default assignment for <code>STAMP</code> as set in the <code>meta/conf/bitbake.conf</code> file is:</p> <pre>STAMP = "\${STAMPS_DIR}/\${MULTIMACH_TARGET_SYS}/\${PN}/\${EXTENDPE}\${PV}-\${PR}</pre> <p>For information on how BitBake uses stamp files to determine if a task should be rerun, see the "<a href="#">Stamp Files and the Rerunning of Tasks</a>" section in the Yocto Project Overview and Concepts Manual.</p> <p>See <code>STAMPS_DIR</code>, <code>MULTIMACH_TARGET_SYS</code>, <code>PN</code>, <code>EXTENDPE</code>, <code>PV</code>, and <code>PR</code> for related variable information.</p>                                             |
| <code>STAMPS_DIR</code>               | Specifies the base directory in which the OpenEmbedded build system places stamps. The default directory is <code>\${TMPDIR}/stamps</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <code>STRIP</code>                    | The minimal command and arguments to run <code>strip</code> , which is used to strip symbols.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <code>SUMMARY</code>                  | The short (72 characters or less) summary of the binary package for packaging systems such as <code>opkg</code> , <code>rpm</code> , or <code>dpkg</code> . By default, <code>SUMMARY</code> is used to define the <code>DESCRIPTION</code> variable if <code>DESCRIPTION</code> is not set in the recipe.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <code>SVN_DIR</code>                  | The directory in which files checked out of a Subversion system are stored.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <code>SYSLINUX_DEFAULT_CONSOLE</code> | <p>Specifies the kernel boot default console. If you want to use a console other than the default, set this variable in your recipe as follows where "X" is the console number you want to use:</p> <pre>SYSLINUX_DEFAULT_CONSOLE = "console=ttyX"</pre> <p>The <code>syslinux</code> class initially sets this variable to null but then checks for a value later.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <code>SYSLINUX_OPTS</code>            | <p>Lists additional options to add to the syslinux file. You need to set this variable in your recipe. If you want to list multiple options, separate the options with a semicolon character (<code>;</code>).</p> <p>The <code>syslinux</code> class uses this variable to create a set of options.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <code>SYSLINUX_SERIAL</code>          | Specifies the alternate serial port or turns it off. To turn off serial, set this variable to an empty string in your recipe. The variable's default value is set in the <code>syslinux</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |

class as follows:

```
SYSINUX_SERIAL ?= "0 115200"
```

The class checks for and uses the variable as needed.

`SYSINUX_SPLASH`

An `.LSS` file used as the background for the VGA boot menu when you use the boot menu. You need to set this variable in your recipe.

The `syslinux` class checks for this variable and if found, the OpenEmbedded build system installs the splash screen.

`SYSINUX_SERIAL_TTY`

Specifies the alternate console=`tty...` kernel boot argument. The variable's default value is set in the `syslinux` class as follows:

```
SYSINUX_SERIAL_TTY ?= "console=ttyS0,115200"
```

The class checks for and uses the variable as needed.

`SYSROOT_DESTDIR`

Points to the temporary directory under the work directory (default `"${WORKDIR}/sysroot-destdir"`) where the files populated into the sysroot are assembled during the `do_populate_sysroot` task.

`SYSROOT_DIRS`

Directories that are staged into the sysroot by the `do_populate_sysroot` task. By default, the following directories are staged:

```
SYSROOT_DIRS = " \
 ${includedir} \
 ${libdir} \
 ${base_libdir} \
 ${nonarch_base_libdir} \
 ${datadir} \
"
```

`SYSROOT_DIRS_BLACKLIST`

Directories that are not staged into the sysroot by the `do_populate_sysroot` task. You can use this variable to exclude certain subdirectories of directories listed in `SYSROOT_DIRS` from staging. By default, the following directories are not staged:

```
SYSROOT_DIRS_BLACKLIST = " \
 ${mandir} \
 ${docdir} \
 ${infodir} \
 ${datadir}/locale \
 ${datadir}/applications \
 ${datadir}/fonts \
 ${datadir}/pixmap \
"
```

`SYSROOT_DIRS_NATIVE`

Extra directories staged into the sysroot by the `do_populate_sysroot` task for `-native` recipes, in addition to those specified in `SYSROOT_DIRS`. By default, the following extra directories are staged:

```
SYSROOT_DIRS_NATIVE = " \
 ${bindir} \
 ${sbindir} \
 ${base_bindir} \
 ${base_sbindir} \
 ${libexecdir} \
 ${sysconfdir} \
 ${localstatedir} \
"
```

## Note

Programs built by `-native` recipes run directly from the sysroot (`STAGING_DIR_NATIVE`), which is why additional directories containing program executables and supporting files need to be staged.

`SYSROOT_PREPROCESS_FUNCS`

A list of functions to execute after files are staged into the sysroot. These functions are usually used to apply additional processing on the staged files, or to stage additional files.

`SYSTEMD_AUTO_ENABLE`

When inheriting the `systemd` class, this variable specifies whether the specified service in `SYSTEMD_SERVICE` should start automatically or not. By default, the service is enabled to automatically start at boot time. The default setting is in the `systemd` class as follows:

```
SYSTEMD_AUTO_ENABLE ??= "enable"
```

You can disable the service by setting the variable to "disable".

`SYSTEMD_BOOT_CFG`

When `EFI_PROVIDER` is set to "systemd-boot", the `SYSTEMD_BOOT_CFG` variable specifies the configuration file that should be used. By default, the `systemd-boot` class sets the `SYSTEMD_BOOT_CFG` as follows:

```
SYSTEMD_BOOT_CFG ?= "${S}/loader.conf"
```

For information on Systemd-boot, see the [Systemd-boot documentation](#).

**SYSTEMD\_BOOT\_ENTRIES**

When [EFI\\_PROVIDER](#) is set to "systemd-boot", the `SYSTEMD_BOOT_ENTRIES` variable specifies a list of entry files (`*.conf`) to install that contain one boot entry per file. By default, the [systemd-boot](#) class sets the `SYSTEMD_BOOT_ENTRIES` as follows:

```
SYSTEMD_BOOT_ENTRIES ?= ""
```

For information on Systemd-boot, see the [Systemd-boot documentation](#).

**SYSTEMD\_BOOT\_TIMEOUT**

When [EFI\\_PROVIDER](#) is set to "systemd-boot", the `SYSTEMD_BOOT_TIMEOUT` variable specifies the boot menu timeout in seconds. By default, the [systemd-boot](#) class sets the `SYSTEMD_BOOT_TIMEOUT` as follows:

```
SYSTEMD_BOOT_TIMEOUT ?= "10"
```

For information on Systemd-boot, see the [Systemd-boot documentation](#).

**SYSTEMD\_PACKAGES**

When inheriting the [systemd](#) class, this variable locates the systemd unit files when they are not found in the main recipe's package. By default, the `SYSTEMD_PACKAGES` variable is set such that the systemd unit files are assumed to reside in the recipes main package:

```
SYSTEMD_PACKAGES ?= "${PN}"
```

If these unit files are not in this recipe's main package, you need to use `SYSTEMD_PACKAGES` to list the package or packages in which the build system can find the systemd unit files.

**SYSTEMD\_SERVICE**

When inheriting the [systemd](#) class, this variable specifies the systemd service name for a package.

When you specify this file in your recipe, use a package name override to indicate the package to which the value applies. Here is an example from the connman recipe:

```
SYSTEMD_SERVICE_${PN} = "connman.service"
```

**SYSVINIT\_ENABLED\_GETTYS**

When using [SysVinit](#), specifies a space-separated list of the virtual terminals that should run a [getty](#) (allowing login), assuming [USE\\_VT](#) is not set to "0".

The default value for `SYSVINIT_ENABLED_GETTYS` is "1" (i.e. only run a getty on the first virtual terminal).

## T

**T**

This variable points to a directory where BitBake places temporary files, which consist mostly of task logs and scripts, when building a particular recipe. The variable is typically set as follows:

```
T = "${WORKDIR}/temp"
```

The [WORKDIR](#) is the directory into which BitBake unpacks and builds the recipe. The default `bitbake.conf` file sets this variable.

The `T` variable is not to be confused with the [TMPDIR](#) variable, which points to the root of the directory tree where BitBake places the output of an entire build.

**TARGET\_ARCH**

The target machine's architecture. The OpenEmbedded build system supports many architectures. Here is an example list of architectures supported. This list is by no means complete as the architecture is configurable:

```
arm
i586
x86_64
powerpc
powerpc64
mips
mipsel
```

For additional information on machine architectures, see the [TUNE\\_ARCH](#) variable.

**TARGET\_AS\_ARCH**

Specifies architecture-specific assembler flags for the target system. `TARGET_AS_ARCH` is initialized from [TUNE\\_ASARGS](#) by default in the BitBake configuration file (`meta/conf/bitbake.conf`):

```
TARGET_AS_ARCH = "${TUNE_ASARGS}"
```

**TARGET\_CC\_ARCH**

Specifies architecture-specific C compiler flags for the target system. `TARGET_CC_ARCH` is initialized from [TUNE\\_CCARGS](#) by default.

## Note

It is a common workaround to append `LDFLAGS` to `TARGET_CC_ARCH` in recipes that build software for the target that would not otherwise respect the exported `LDFLAGS` variable.

|                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>TARGET_CC_KERNEL_ARCH</code> | This is a specific kernel compiler flag for a CPU or Application Binary Interface (ABI) tune. The flag is used rarely and only for cases where a userspace <code>TUNE_CCARGS</code> is not compatible with the kernel compilation. The <code>TARGET_CC_KERNEL_ARCH</code> variable allows the kernel (and associated modules) to use a different configuration. See the <code>meta/conf/machine/include/arm/feature-arm-thumb.inc</code> file in the <a href="#">Source Directory</a> for an example.                                                                   |
| <code>TARGET_CFLAGS</code>         | Specifies the flags to pass to the C compiler when building for the target. When building in the target context, <code>CFLAGS</code> is set to the value of this variable by default.<br><br>Additionally, the SDK's environment setup script sets the <code>CFLAGS</code> variable in the environment to the <code>TARGET_CFLAGS</code> value so that executables built using the SDK also have the flags applied.                                                                                                                                                     |
| <code>TARGET_CPPFLAGS</code>       | Specifies the flags to pass to the C pre-processor (i.e. to both the C and the C++ compilers) when building for the target. When building in the target context, <code>CPPFLAGS</code> is set to the value of this variable by default.<br><br>Additionally, the SDK's environment setup script sets the <code>CPPFLAGS</code> variable in the environment to the <code>TARGET_CPPFLAGS</code> value so that executables built using the SDK also have the flags applied.                                                                                               |
| <code>TARGET_CXXFLAGS</code>       | Specifies the flags to pass to the C++ compiler when building for the target. When building in the target context, <code>CXXFLAGS</code> is set to the value of this variable by default.<br><br>Additionally, the SDK's environment setup script sets the <code>CXXFLAGS</code> variable in the environment to the <code>TARGET_CXXFLAGS</code> value so that executables built using the SDK also have the flags applied.                                                                                                                                             |
| <code>TARGET_FPU</code>            | Specifies the method for handling FPU code. For FPU-less targets, which include most ARM CPUs, the variable must be set to "soft". If not, the kernel emulation gets used, which results in a performance penalty.                                                                                                                                                                                                                                                                                                                                                      |
| <code>TARGET_LD_ARCH</code>        | Specifies architecture-specific linker flags for the target system. <code>TARGET_LD_ARCH</code> is initialized from <code>TUNE_LDARGS</code> by default in the BitBake configuration file ( <code>meta/conf/bitbake.conf</code> ):<br><br><pre>TARGET_LD_ARCH = "\${TUNE_LDARGS}"</pre>                                                                                                                                                                                                                                                                                 |
| <code>TARGET_LDFLAGS</code>        | Specifies the flags to pass to the linker when building for the target. When building in the target context, <code>LDFLAGS</code> is set to the value of this variable by default.<br><br>Additionally, the SDK's environment setup script sets the <code>LDFLAGS</code> variable in the environment to the <code>TARGET_LDFLAGS</code> value so that executables built using the SDK also have the flags applied.                                                                                                                                                      |
| <code>TARGET_OS</code>             | Specifies the target's operating system. The variable can be set to "linux" for glibc-based systems (GNU C Library) and to "linux-musl" for musl libc. For ARM/EABI targets, "linux-gnueabi" and "linux-musleabi" possible values exist.                                                                                                                                                                                                                                                                                                                                |
| <code>TARGET_PREFIX</code>         | Specifies the prefix used for the toolchain binary target tools.<br><br>Depending on the type of recipe and the build target, <code>TARGET_PREFIX</code> is set as follows: <ul style="list-style-type: none"> <li>For recipes building for the target machine, the value is <code>"\${TARGET_SYS}"</code>.</li> <li>For native recipes, the build system sets the variable to the value of <code>BUILD_PREFIX</code>.</li> <li>For native SDK recipes (<code>nativesdk</code>), the build system sets the variable to the value of <code>SDK_PREFIX</code>.</li> </ul> |
| <code>TARGET_SYS</code>            | Specifies the system, including the architecture and the operating system, for which the build is occurring in the context of the current recipe.<br><br>The OpenEmbedded build system automatically sets this variable based on <code>TARGET_ARCH</code> , <code>TARGET_VENDOR</code> , and <code>TARGET_OS</code> variables.                                                                                                                                                                                                                                          |

## Note

You do not need to set the `TARGET_SYS` variable yourself.

Consider these two examples:

- Given a native recipe on a 32-bit, x86 machine running Linux, the value is "i686-linux".
- Given a recipe being built for a little-endian, MIPS target running Linux, the value might be "mipsel-linux".

`TARGET_VENDOR`

Specifies the name of the target vendor.

`TCLIBC`

Specifies the GNU standard C library (`libc`) variant to use during the build process. This variable replaces `POKYLIBC`, which is no longer supported.

You can select "glibc", "musl", "newlib", or "baremetal"

`TCLIBCAPPEND`

Specifies a suffix to be appended onto the `TMPDIR` value. The suffix identifies the `libc` variant for building. When you are building for multiple variants with the same Build Directory, this mechanism ensures that output for different `libc` variants is kept separate to avoid potential conflicts.

In the `defaultsetup.conf` file, the default value of `TCLIBCAPPEND` is "-\${TCLIBC}". However, distros such as poky, which normally only support one `libc` variant, set `TCLIBCAPPEND` to "" in their distro configuration file resulting in no suffix being applied.

`TCMODE`

Specifies the toolchain selector. `TCMODE` controls the characteristics of the generated packages and images by telling the OpenEmbedded build system which toolchain profile to use. By default, the OpenEmbedded build system builds its own internal toolchain. The variable's default value is "default", which uses that internal toolchain.

## Note

If `TCMODE` is set to a value other than "default", then it is your responsibility to ensure that the toolchain is compatible with the default toolchain. Using older or newer versions of these components might cause build problems. See the Release Notes for the Yocto Project release for the specific components with which the toolchain must be compatible. To access the Release Notes, go to the [Downloads](#) page on the Yocto Project website and click on the "RELEASE INFORMATION" link for the appropriate release.

The `TCMODE` variable is similar to `TCLIBC`, which controls the variant of the GNU standard C library (`libc`) used during the build process: `glibc` or `musl`.

With additional layers, it is possible to use a pre-compiled external toolchain. One example is the Sourcery G++ Toolchain. The support for this toolchain resides in the separate Mentor Graphics® `meta-sourcery` layer at <http://github.com/MentorEmbedded/meta-sourcery/>.

The layer's `README` file contains information on how to use the Sourcery G++ Toolchain as an external toolchain. In summary, you must be sure to add the layer to your `bblayers.conf` file in front of the `meta` layer and then set the `EXTERNAL_TOOLCHAIN` variable in your `local.conf` file to the location in which you installed the toolchain.

The fundamentals used for this example apply to any external toolchain. You can use `meta-sourcery` as a template for adding support for other external toolchains.

`TEST_EXPORT_DIR`

The location the OpenEmbedded build system uses to export tests when the `TEST_EXPORT_ONLY` variable is set to "1".

The `TEST_EXPORT_DIR` variable defaults to "`${TMPDIR}/testimage/${PN}`".

`TEST_EXPORT_ONLY`

Specifies to export the tests only. Set this variable to "1" if you do not want to run the tests but you want them to be exported in a manner that you to run them outside of the build system.

`TEST_LOG_DIR`

Holds the SSH log and the boot log for QEMU machines. The `TEST_LOG_DIR` variable defaults to "`${WORKDIR}/testimage`".

## Note

Actual test results reside in the task log (`log.do_testimage`), which is in the `${WORKDIR}/temp/` directory.

`TEST_POWERCONTROL_CMD`

For automated hardware testing, specifies the command to use to control the power of the target machine under test. Typically, this command would point to a script that performs the appropriate action (e.g. interacting with a web-enabled power strip). The specified command should expect to receive as the last argument "off", "on" or "cycle"

specifying to power off, on, or cycle (power off and then power on) the device, respectively.

`TEST_POWERCONTROL_EXTRA_ARGS¶`

For automated hardware testing, specifies additional arguments to pass through to the command specified in `TEST_POWERCONTROL_CMD`. Setting `TEST_POWERCONTROL_EXTRA_ARGS` is optional. You can use it if you wish, for example, to separate the machine-specific and non-machine-specific parts of the arguments.

`TEST_QEMUBOOT_TIMEOUT¶`

The time in seconds allowed for an image to boot before automated runtime tests begin to run against an image. The default timeout period to allow the boot process to reach the login prompt is 500 seconds. You can specify a different value in the `local.conf` file.

For more information on testing images, see the "[Performing Automated Runtime Testing](#)" section in the Yocto Project Development Tasks Manual.

`TEST_SERIALCONTROL_CMD¶`

For automated hardware testing, specifies the command to use to connect to the serial console of the target machine under test. This command simply needs to connect to the serial console and forward that connection to standard input and output as any normal terminal program does.

For example, to use the Picocom terminal program on serial device `/dev/ttyUSB0` at 115200bps, you would set the variable as follows:

```
TEST_SERIALCONTROL_CMD = "picocom /dev/ttyUSB0 -b 115200"
```

`TEST_SERIALCONTROL_EXTRA_ARGS¶`

For automated hardware testing, specifies additional arguments to pass through to the command specified in `TEST_SERIALCONTROL_CMD`. Setting `TEST_SERIALCONTROL_EXTRA_ARGS` is optional. You can use it if you wish, for example, to separate the machine-specific and non-machine-specific parts of the command.

`TEST_SERVER_IP¶`

The IP address of the build machine (host machine). This IP address is usually automatically detected. However, if detection fails, this variable needs to be set to the IP address of the build machine (i.e. where the build is taking place).

## Note

The `TEST_SERVER_IP` variable is only used for a small number of tests such as the "dnf" test suite, which needs to download packages from `WORKDIR/oe-rootfs-repo`.

`TEST_TARGET¶`

Specifies the target controller to use when running tests against a test image. The default controller to use is "QemuTarget":

```
TEST_TARGET = "QemuTarget"
```

A target controller is a class that defines how an image gets deployed on a target and how a target is started. A layer can extend the controllers by adding a module in the layer's `/lib/oeqa/controllers` directory and by inheriting the `BaseTarget` class, which is an abstract class that cannot be used as a value of `TEST_TARGET`.

You can provide the following arguments with `TEST_TARGET`:

- **"QemuTarget"**: Boots a QEMU image and runs the tests. See the "[Enabling Runtime Tests on QEMU](#)" section in the Yocto Project Development Tasks Manual for more information.
- **"SimpleRemoteTarget"**: Runs the tests on target hardware that is already up and running. The hardware can be on the network or it can be a device running an image on QEMU. You must also set `TEST_TARGET_IP` when you use "SimpleRemoteTarget".

## Note

This argument is defined in `meta/lib/oeqa/controllers/simpleremote.py`.

For information on running tests on hardware, see the "[Enabling Runtime Tests on Hardware](#)" section in the Yocto Project Development Tasks Manual.

`TEST_TARGET_IP¶`

The IP address of your hardware under test. The `TEST_TARGET_IP` variable has no effect when `TEST_TARGET` is set to "qemu".

When you specify the IP address, you can also include a port. Here is an example:

```
TEST_TARGET_IP = "192.168.1.4:2201"
```

Specifying a port is useful when SSH is started on a non-standard port or in cases when your hardware under test is behind a firewall or network that is not directly accessible from your host and you need to do port address translation.

#### TEST\_SUITES¶

An ordered list of tests (modules) to run against an image when performing automated runtime testing.

The OpenEmbedded build system provides a core set of tests that can be used against images.

### Note

Currently, there is only support for running these tests under QEMU.

Tests include `ping`, `ssh`, `df` among others. You can add your own tests to the list of tests by appending `TEST_SUITES` as follows:

```
TEST_SUITES_append = " mytest"
```

Alternatively, you can provide the "auto" option to have all applicable tests run against the image.

```
TEST_SUITES_append = " auto"
```

Using this option causes the build system to automatically run tests that are applicable to the image. Tests that are not applicable are skipped.

The order in which tests are run is important. Tests that depend on another test must appear later in the list than the test on which they depend. For example, if you append the list of tests with two tests (`test_A` and `test_B`) where `test_B` is dependent on `test_A`, then you must order the tests as follows:

```
TEST_SUITES = " test_A test_B"
```

For more information on testing images, see the "[Performing Automated Runtime Testing](#)" section in the Yocto Project Development Tasks Manual.

#### TESTIMAGE\_AUTO¶

Automatically runs the series of automated tests for images when an image is successfully built. Setting `TESTIMAGE_AUTO` to "1" causes any image that successfully builds to automatically boot under QEMU. Using the variable also adds in dependencies so that any SDK for which testing is requested is automatically built first.

These tests are written in Python making use of the `unittest` module, and the majority of them run commands on the target system over `ssh`. You can set this variable to "1" in your `local.conf` file in the [Build Directory](#) to have the OpenEmbedded build system automatically run these tests after an image successfully builds:

```
TESTIMAGE_AUTO = "1"
```

For more information on enabling, running, and writing these tests, see the "[Performing Automated Runtime Testing](#)" section in the Yocto Project Development Tasks Manual and the "[testimage\\*.bbclass](#)" section.

#### THISDIR¶

The directory in which the file BitBake is currently parsing is located. Do not manually set this variable.

#### TIME¶

The time the build was started. Times appear using the hour, minute, and second (HMS) format (e.g. "140159" for one minute and fifty-nine seconds past 1400 hours).

#### TMPDIR¶

This variable is the base directory the OpenEmbedded build system uses for all build output and intermediate files (other than the shared state cache). By default, the `TMPDIR` variable points to `tmp` within the [Build Directory](#).

If you want to establish this directory in a location other than the default, you can uncomment and edit the following statement in the `conf/local.conf` file in the [Source Directory](#):

```
#TMPDIR = "${TOPDIR}/tmp"
```

An example use for this scenario is to set `TMPDIR` to a local disk, which does not use NFS, while having the Build Directory use NFS.

The filesystem used by `TMPDIR` must have standard filesystem semantics (i.e. mixed-case files are unique, POSIX file locking, and persistent inodes). Due to various issues with NFS and bugs in some implementations, NFS does not meet this minimum requirement. Consequently, `TMPDIR` cannot be on NFS.

#### TOOLCHAIN\_HOST\_TASK¶

This variable lists packages the OpenEmbedded build system uses when building an SDK, which contains a cross-development environment. The packages specified by this variable are part of the toolchain set that runs on the [SDKMACHINE](#), and each package should usually have the prefix `nativesdk-`. For example, consider the following command when building an SDK:



```
$ bitbake -c populate_sdk imagename
```

In this case, a default list of packages is set in this variable, but you can add additional packages to the list. See the "[Adding Individual Packages to the Standard SDK](#)" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual for more information.

For background information on cross-development toolchains in the Yocto Project development environment, see the "[Cross-Development Toolchain Generation](#)" section in the Yocto Project Overview and Concepts Manual. For information on setting up a cross-development environment, see the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

#### TOOLCHAIN\_OUTPUTNAME¶

This variable defines the name used for the toolchain output. The `populate_sdk_base` class sets the `TOOLCHAIN_OUTPUTNAME` variable as follows:

```
TOOLCHAIN_OUTPUTNAME ?= "${SDK_NAME}-toolchain-${SDK_VERSION}"
```

See the `SDK_NAME` and `SDK_VERSION` variables for additional information.

#### TOOLCHAIN\_TARGET\_TASK¶

This variable lists packages the OpenEmbedded build system uses when it creates the target part of an SDK (i.e. the part built for the target hardware), which includes libraries and headers. Use this variable to add individual packages to the part of the SDK that runs on the target. See the "[Adding Individual Packages to the Standard SDK](#)" section in the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual for more information.

For background information on cross-development toolchains in the Yocto Project development environment, see the "[Cross-Development Toolchain Generation](#)" section in the Yocto Project Overview and Concepts Manual. For information on setting up a cross-development environment, see the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual.

#### TOPDIR¶

The top-level [Build Directory](#). BitBake automatically sets this variable when you initialize your build environment using `oe-init-build-env`.

#### TRANSLATED\_TARGET\_ARCH¶

A sanitized version of `TARGET_ARCH`. This variable is used where the architecture is needed in a value where underscores are not allowed, for example within package filenames. In this case, dash characters replace any underscore characters used in `TARGET_ARCH`.

Do not edit this variable.

#### TUNE\_ARCH¶

The GNU canonical architecture for a specific architecture (i.e. `arm`, `armeb`, `mips`, `mips64`, and so forth). BitBake uses this value to setup configuration.

`TUNE_ARCH` definitions are specific to a given architecture. The definitions can be a single static definition, or can be dynamically adjusted. You can see details for a given CPU family by looking at the architecture's `README` file. For example, the `meta/conf/machine/include/mips/README` file in the [Source Directory](#), provides information for `TUNE_ARCH` specific to the `mips` architecture.

`TUNE_ARCH` is tied closely to `TARGET_ARCH`, which defines the target machine's architecture. The BitBake configuration file (`meta/conf/bitbake.conf`) sets `TARGET_ARCH` as follows:

```
TARGET_ARCH = "${TUNE_ARCH}"
```

The following list, which is by no means complete since architectures are configurable, shows supported machine architectures:

```
arm
i586
x86_64
powerpc
powerpc64
mips
mipsel
```

#### TUNE\_ASARGS¶

Specifies architecture-specific assembler flags for the target system. The set of flags is based on the selected tune features. `TUNE_ASARGS` is set using the tune include files, which are typically under `meta/conf/machine/include/` and are influenced through `TUNE_FEATURES`. For example, the `meta/conf/machine/include/x86/arch-x86.inc` file defines the flags for the `x86` architecture as follows:

```
TUNE_ASARGS += "${@bb.utils.contains("TUNE_FEATURES", "mx32", "-x32", "", d
```



### Note

Board Support Packages (BSPs) select the tune. The selected tune, in turn, affects the tune variables themselves (i.e. the tune can supply its own set of flags).

**TUNE\_CCARGS**

Specifies architecture-specific C compiler flags for the target system. The set of flags is based on the selected tune features. `TUNE_CCARGS` is set using the tune include files, which are typically under `meta/conf/machine/include/` and are influenced through [TUNE\\_FEATURES](#).

**Note**

Board Support Packages (BSPs) select the tune. The selected tune, in turn, affects the tune variables themselves (i.e. the tune can supply its own set of flags).

**TUNE\_LDARGS**

Specifies architecture-specific linker flags for the target system. The set of flags is based on the selected tune features. `TUNE_LDARGS` is set using the tune include files, which are typically under `meta/conf/machine/include/` and are influenced through [TUNE\\_FEATURES](#). For example, the `meta/conf/machine/include/x86/arch-x86.inc` file defines the flags for the x86 architecture as follows:

```
TUNE_LDARGS += "${@bb.utils.contains("TUNE_FEATURES", "mx32", "-m elf32_x86
```

**Note**

Board Support Packages (BSPs) select the tune. The selected tune, in turn, affects the tune variables themselves (i.e. the tune can supply its own set of flags).

**TUNE\_FEATURES**

Features used to "tune" a compiler for optimal use given a specific processor. The features are defined within the tune files and allow arguments (i.e. `TUNE_*ARGS`) to be dynamically generated based on the features.

The OpenEmbedded build system verifies the features to be sure they are not conflicting and that they are supported.

The BitBake configuration file (`meta/conf/bitbake.conf`) defines `TUNE_FEATURES` as follows:

```
TUNE_FEATURES ??= "${TUNE_FEATURES_tune-${DEFAULTTUNE}}"
```

See the [DEFAULTTUNE](#) variable for more information.

**TUNE\_PKGARCH**

The package architecture understood by the packaging system to define the architecture, ABI, and tuning of output packages. The specific tune is defined using the `"_tune"` override as follows:

```
TUNE_PKGARCH_tune-tune = "tune"
```

These tune-specific package architectures are defined in the machine include files. Here is an example of the "core2-32" tuning as used in the `meta/conf/machine/include/tune-core2.inc` file:

```
TUNE_PKGARCH_tune-core2-32 = "core2-32"
```

**TUNEABI**

An underlying Application Binary Interface (ABI) used by a particular tuning in a given toolchain layer. Providers that use prebuilt libraries can use the `TUNEABI`, [TUNEABI\\_OVERRIDE](#), and [TUNEABI\\_WHITELIST](#) variables to check compatibility of tunings against their selection of libraries.

If `TUNEABI` is undefined, then every tuning is allowed. See the [sanity](#) class to see how the variable is used.

**TUNEABI\_OVERRIDE**

If set, the OpenEmbedded system ignores the [TUNEABI\\_WHITELIST](#) variable. Providers that use prebuilt libraries can use the `TUNEABI_OVERRIDE`, `TUNEABI_WHITELIST`, and [TUNEABI](#) variables to check compatibility of a tuning against their selection of libraries.

See the [sanity](#) class to see how the variable is used.

**TUNEABI\_WHITELIST**

A whitelist of permissible [TUNEABI](#) values. If `TUNEABI_WHITELIST` is not set, all tunes are allowed. Providers that use prebuilt libraries can use the `TUNEABI_WHITELIST`, [TUNEABI\\_OVERRIDE](#), and `TUNEABI` variables to check compatibility of a tuning against their selection of libraries.

See the [sanity](#) class to see how the variable is used.

**TUNECONFLICTS[*feature*]**

Specifies CPU or Application Binary Interface (ABI) tuning features that conflict with *feature*.

Known tuning conflicts are specified in the machine include files in the [Source Directory](#). Here is an example from the `meta/conf/machine/include/mips/arch-mips.inc` include file that lists the "o32" and "n64" features as conflicting with the "n32" feature:

```
TUNECONFLICTS[n32] = "o32 n64"
```

`TUNEVALID[feature]`

Specifies a valid CPU or Application Binary Interface (ABI) tuning feature. The specified feature is stored as a flag. Valid features are specified in the machine include files (e.g. `meta/conf/machine/include/arm/arch-arm.inc`). Here is an example from that file:

```
TUNEVALID[bigendian] = "Enable big-endian mode."
```

See the machine include files in the [Source Directory](#) for these features.

## U

`UBOOT_CONFIG`

Configures the `UBOOT_MACHINE` and can also define `IMAGE_FSTYPES` for individual cases.

Following is an example from the `meta-fsl-arm` layer.

```
UBOOT_CONFIG ??= "sd"
UBOOT_CONFIG[sd] = "mx6qsabreauto_config,sdcard"
UBOOT_CONFIG[eimnor] = "mx6qsabreauto_eimnor_config"
UBOOT_CONFIG[nand] = "mx6qsabreauto_nand_config,ubifs"
UBOOT_CONFIG[spinor] = "mx6qsabreauto_spinor_config"
```

In this example, "sd" is selected as the configuration of the possible four for the `UBOOT_MACHINE`. The "sd" configuration defines "mx6qsabreauto\_config" as the value for `UBOOT_MACHINE`, while the "sdcard" specifies the `IMAGE_FSTYPES` to use for the U-boot image.

For more information on how the `UBOOT_CONFIG` is handled, see the [uboot-config](#) class.

`UBOOT_ENTRYPOINT`

Specifies the entry point for the U-Boot image. During U-Boot image creation, the `UBOOT_ENTRYPOINT` variable is passed as a command-line parameter to the `uboot-mkimage` utility.

`UBOOT_LOADADDRESS`

Specifies the load address for the U-Boot image. During U-Boot image creation, the `UBOOT_LOADADDRESS` variable is passed as a command-line parameter to the `uboot-mkimage` utility.

`UBOOT_LOCALVERSION`

Appends a string to the name of the local version of the U-Boot image. For example, assuming the version of the U-Boot image built was "2013.10", the full version string reported by U-Boot would be "2013.10-yocto" given the following statement:

```
UBOOT_LOCALVERSION = "-yocto"
```

`UBOOT_MACHINE`

Specifies the value passed on the `make` command line when building a U-Boot image. The value indicates the target platform configuration. You typically set this variable from the machine configuration file (i.e. `conf/machine/machine_name.conf`).

Please see the "Selection of Processor Architecture and Board Type" section in the U-Boot README for valid values for this variable.

`UBOOT_MAKE_TARGET`

Specifies the target called in the `Makefile`. The default target is "all".

`UBOOT_SUFFIX`

Points to the generated U-Boot extension. For example, `u-boot.sb` has a `.sb` extension.

The default U-Boot extension is `.bin`

`UBOOT_TARGET`

Specifies the target used for building U-Boot. The target is passed directly as part of the "make" command (e.g. SPL and AIS). If you do not specifically set this variable, the OpenEmbedded build process passes and uses "all" for the target during the U-Boot building process.

`UNKNOWN_CONFIGURE_WHITELIST`

Specifies a list of options that, if reported by the configure script as being invalid, should not generate a warning during the [do\\_configure](#) task. Normally, invalid configure options are simply not passed to the configure script (e.g. should be removed from [EXTRA\\_OECONF](#) or [PACKAGECONFIG\\_CONFARGS](#)). However, common options, for example, exist that are passed to all configure scripts at a class level that might not be valid for some configure scripts. It follows that no benefit exists in seeing a warning about these options. For these cases, the options are added to `UNKNOWN_CONFIGURE_WHITELIST`.

The configure arguments check that uses `UNKNOWN_CONFIGURE_WHITELIST` is part of the [insane](#) class and is only enabled if the recipe inherits the [autotools](#) class.

`UPDATERCPN`

For recipes inheriting the [update-rc.d](#) class, `UPDATERCPN` specifies the package that contains the `initscript` that is enabled.

The default value is "\${PN}". Given that almost all recipes that install `initscripts` package them in the main package for the recipe, you rarely need to set this variable in individual recipes.

`UPSTREAM_CHECK_GITTAGREGEX`

You can perform a per-recipe check for what the latest upstream source code version is by calling `bitbake -c checkpkg recipe`. If the recipe source code is provided from Git repositories, the OpenEmbedded build system determines the latest upstream version by picking the latest tag from the list of all repository tags.

You can use the `UPSTREAM_CHECK_GITTAGREGEX` variable to provide a regular expression to filter only the relevant tags should the default filter not work correctly.

```
UPSTREAM_CHECK_GITTAGREGEX = "git_tag_regex"
```

#### UPSTREAM\_CHECK\_REGEX¶

Use the `UPSTREAM_CHECK_REGEX` variable to specify a different regular expression instead of the default one when the package checking system is parsing the page found using `UPSTREAM_CHECK_URI`.

```
UPSTREAM_CHECK_REGEX = "package_regex"
```

#### UPSTREAM\_CHECK\_URI¶

You can perform a per-recipe check for what the latest upstream source code version is by calling `bitbake -c checkpkg recipe`. If the source code is provided from tarballs, the latest version is determined by fetching the directory listing where the tarball is and attempting to find a later tarball. When this approach does not work, you can use `UPSTREAM_CHECK_URI` to provide a different URI that contains the link to the latest tarball.

```
UPSTREAM_CHECK_URI = "recipe_url"
```

#### USE\_DEVFS¶

Determines if `devtmpfs` is used for `/dev` population. The default value used for `USE_DEVFS` is "1" when no value is specifically set. Typically, you would set `USE_DEVFS` to "0" for a statically populated `/dev` directory.

See the "[Selecting a Device Manager](#)" section in the Yocto Project Development Tasks Manual for information on how to use this variable.

#### USE\_VT¶

When using `SysVinit`, determines whether or not to run a `getty` on any virtual terminals in order to enable logging in through those terminals.

The default value used for `USE_VT` is "1" when no default value is specifically set. Typically, you would set `USE_VT` to "0" in the machine configuration file for machines that do not have a graphical display attached and therefore do not need virtual terminal functionality.

#### USER\_CLASSES¶

A list of classes to globally inherit. These classes are used by the OpenEmbedded build system to enable extra features (e.g. `buildstats`, `image-mklibs`, and so forth).

The default list is set in your `local.conf` file:

```
USER_CLASSES ?= "buildstats image-mklibs image-prelink"
```

For more information, see `meta-poky/conf/local.conf.sample` in the [Source Directory](#).

#### USERADD\_ERROR\_DYNAMIC¶

If set to "error", forces the OpenEmbedded build system to produce an error if the user identification (`uid`) and group identification (`gid`) values are not defined in `files/passwd` and `files/group` files. If set to "warn", a warning will be issued instead.

The default behavior for the build system is to dynamically apply `uid` and `gid` values. Consequently, the `USERADD_ERROR_DYNAMIC` variable is by default not set. If you plan on using statically assigned `gid` and `uid` values, you should set the `USERADD_ERROR_DYNAMIC` variable in your `local.conf` file as follows:

```
USERADD_ERROR_DYNAMIC = "error"
```

Overriding the default behavior implies you are going to also take steps to set static `uid` and `gid` values through use of the `USERADDEXTENSION`, `USERADD_UID_TABLES`, and `USERADD_GID_TABLES` variables.

#### USERADD\_GID\_TABLES¶

Specifies a password file to use for obtaining static group identification (`gid`) values when the OpenEmbedded build system adds a group to the system during package installation.

When applying static group identification (`gid`) values, the OpenEmbedded build system looks in `BBPATH` for a `files/group` file and then applies those `uid` values. Set the variable as follows in your `local.conf` file:

```
USERADD_GID_TABLES = "files/group"
```

## Note

Setting the `USERADDEXTENSION` variable to "useradd-staticids" causes the build system to use static `gid` values.

**USERADD\_PACKAGES¶**

When inheriting the `useradd` class, this variable specifies the individual packages within the recipe that require users and/or groups to be added.

You must set this variable if the recipe inherits the class. For example, the following enables adding a user for the main package in a recipe:

```
USERADD_PACKAGES = "${PN}"
```

**Note**

It follows that if you are going to use the `USERADD_PACKAGES` variable, you need to set one or more of the `USERADD_PARAM`, `GROUPADD_PARAM`, or `GROUPMEMS_PARAM` variables.

**USERADD\_PARAM¶**

When inheriting the `useradd` class, this variable specifies for a package what parameters should pass to the `useradd` command if you add a user to the system when the package is installed.

Here is an example from the `dbus` recipe:

```
USERADD_PARAM_${PN} = "--system --home ${localstatedir}/lib/dbus \
 --no-create-home --shell /bin/false \
 --user-group messagebus"
```

For information on the standard Linux shell command `useradd`, see <http://linux.die.net/man/8/useradd>.

**USERADD\_UID\_TABLES¶**

Specifies a password file to use for obtaining static user identification (`uid`) values when the OpenEmbedded build system adds a user to the system during package installation.

When applying static user identification (`uid`) values, the OpenEmbedded build system looks in `BBPATH` for a `files/passwd` file and then applies those `uid` values. Set the variable as follows in your `local.conf` file:

```
USERADD_UID_TABLES = "files/passwd"
```

**Note**

Setting the `USERADDEXTENSION` variable to "useradd-staticids" causes the build system to use static `uid` values.

**USERADDEXTENSION¶**

When set to "useradd-staticids", causes the OpenEmbedded build system to base all user and group additions on a static `passwd` and `group` files found in `BBPATH`.

To use static user identification (`uid`) and group identification (`gid`) values, set the variable as follows in your `local.conf` file:

```
USERADDEXTENSION = "useradd-staticids"
```

**Note**

Setting this variable to use static `uid` and `gid` values causes the OpenEmbedded build system to employ the `useradd-staticids` class.

If you use static `uid` and `gid` information, you must also specify the `files/passwd` and `files/group` files by setting the `USERADD_UID_TABLES` and `USERADD_GID_TABLES` variables. Additionally, you should also set the `USERADD_ERROR_DYNAMIC` variable.

**V****VOLATILE\_LOG\_DIR¶**

Specifies the persistence of the target's `/var/log` directory, which is used to house postinstall target log files.

By default, `VOLATILE_LOG_DIR` is set to "yes", which means the file is not persistent. You can override this setting by setting the variable to "no" to make the log directory persistent.

**W**

**WARN\_QA¶**

Specifies the quality assurance checks whose failures are reported as warnings by the OpenEmbedded build system. You set this variable in your distribution configuration file. For a list of the checks you can control with this variable, see the "[insane.bbclass](#)" section.

**WKS\_FILE\_DEPENDS¶**

When placed in the recipe that builds your image, this variable lists build-time dependencies. The `WKS_FILE_DEPENDS` variable is only applicable when Wic images are active (i.e. when `IMAGE_FSTYPES` contains entries related to Wic). If your recipe does not create Wic images, the variable has no effect.

The `WKS_FILE_DEPENDS` variable is similar to the `DEPENDS` variable. When you use the variable in your recipe that builds the Wic image, dependencies you list in the `WKS_FILE_DEPENDS` variable are added to the `DEPENDS` variable.

With the `WKS_FILE_DEPENDS` variable, you have the possibility to specify a list of additional dependencies (e.g. native tools, bootloaders, and so forth), that are required to build Wic images. Following is an example:

```
WKS_FILE_DEPENDS = "some-native-tool"
```

In the previous example, *some-native-tool* would be replaced with an actual native tool on which the build would depend.

**WKS\_FILE¶**

Specifies the location of the Wic kickstart file that is used by the OpenEmbedded build system to create a partitioned image (*image.wic*). For information on how to create a partitioned image, see the "[Creating Partitioned Images Using Wic](#)" section in the Yocto Project Development Tasks Manual. For details on the kickstart file format, see the "[OpenEmbedded Kickstart \(.wks\)](#) Reference Chapter.

**WORKDIR¶**

The pathname of the work directory in which the OpenEmbedded build system builds a recipe. This directory is located within the `TMPDIR` directory structure and is specific to the recipe being built and the system for which it is being built.

The `WORKDIR` directory is defined as follows:

```
${TMPDIR}/work/${MULTIMACH_TARGET_SYS}/${PN}/${EXTENDPE}${PV}-${PR}
```

The actual directory depends on several things:

- `TMPDIR`: The top-level build output directory
- `MULTIMACH_TARGET_SYS`: The target system identifier
- `PN`: The recipe name
- `EXTENDPE`: The epoch - (if `PE` is not specified, which is usually the case for most recipes, then `EXTENDPE` is blank)
- `PV`: The recipe version
- `PR`: The recipe revision

As an example, assume a Source Directory top-level folder name `poky`, a default Build Directory at `poky/build`, and a `qemux86-poky-linux` machine target system. Furthermore, suppose your recipe is named `foo_1.3.0-r0.bb`. In this case, the work directory the build system uses to build the package would be as follows:

```
poky/build/tmp/work/qemux86-poky-linux/foo/1.3.0-r0
```

**X****XSERVER¶**

Specifies the packages that should be installed to provide an X server and drivers for the current machine, assuming your image directly includes `packagegroup-core-x11-xserver` or, perhaps indirectly, includes "x11-base" in `IMAGE_FEATURES`.

The default value of `XSERVER`, if not specified in the machine configuration, is "xserver-xorg xf86-video-fbdev xf86-input-evdev".

**Chapter 34. Variable Context¶****34.1. Configuration****34.1.1. Distribution (Distro)****34.1.2. Machine****34.1.3. Local****34.2. Recipes****34.2.1. Required****34.2.2. Dependencies****34.2.3. Paths****34.2.4. Extra Build Information**

While you can use most variables in almost any context such as `.conf`, `.bbclass`, `.inc`, and `.bb` files, some variables are often associated with a particular locality or context. This chapter describes some common associations.

**34.1. Configuration¶**

The following subsections provide lists of variables whose context is configuration: distribution, machine, and local.

### 34.1.1. Distribution (Distro)¶¶

This section lists variables whose configuration context is the distribution, or distro.

- DISTRO
- DISTRO\_NAME
- DISTRO\_VERSION
- MAINTAINER
- PACKAGE\_CLASSES
- TARGET\_OS
- TARGET\_FPU
- TCMODE
- TCLIBC

### 34.1.2. Machine¶¶

This section lists variables whose configuration context is the machine.

- TARGET\_ARCH
- SERIAL\_CONSOLES
- PACKAGE\_EXTRA\_ARCHS
- IMAGE\_FSTYPES
- MACHINE\_FEATURES
- MACHINE\_EXTRA\_RDEPENDS
- MACHINE\_EXTRA\_RRECOMMENDS
- MACHINE\_ESSENTIAL\_EXTRA\_RDEPENDS
- MACHINE\_ESSENTIAL\_EXTRA\_RRECOMMENDS

### 34.1.3. Local¶¶

This section lists variables whose configuration context is the local configuration through the `local.conf` file.

- DISTRO
- MACHINE
- DL\_DIR
- BBFILES
- EXTRA\_IMAGE\_FEATURES
- PACKAGE\_CLASSES
- BB\_NUMBER\_THREADS
- BBINCLUDELOGS
- ENABLE\_BINARY\_LOCALE\_GENERATION

## 34.2. Recipes¶¶

The following subsections provide lists of variables whose context is recipes: required, dependencies, path, and extra build information.

### 34.2.1. Required¶¶

This section lists variables that are required for recipes.

- LICENSE
- LIC\_FILES\_CHKSUM
- SRC\_URI - used in recipes that fetch local or remote files.

### 34.2.2. Dependencies¶¶

This section lists variables that define recipe dependencies.

- DEPENDS
- RDEPENDS
- RRECOMMENDS

- [RCONFLICTS](#)
- [REPLACES](#)

### 34.2.3. Paths¶

This section lists variables that define recipe paths.

- [WORKDIR](#)
- [S](#)
- [FILES](#)

### 34.2.4. Extra Build Information¶

This section lists variables that define extra build information for recipes.

- [DEFAULT\\_PREFERENCE](#)
- [EXTRA\\_OECMAKE](#)
- [EXTRA\\_OECONF](#)
- [EXTRA\\_OEMAKE](#)
- [PACKAGECONFIG\\_CONFARGS](#)
- [PACKAGES](#)

## Chapter 35. FAQ¶

### 35.1. How does Poky differ from [OpenEmbedded](#)?

The term "Poky" refers to the specific reference build system that the Yocto Project provides. Poky is based on [OE-Core](#) and [BitBake](#). Thus, the generic term used here for the build system is the "OpenEmbedded build system." Development in the Yocto Project using Poky is closely tied to OpenEmbedded, with changes always being merged to OE-Core or BitBake first before being pulled back into Poky. This practice benefits both projects immediately.

### 35.2. My development system does not meet the required Git, tar, and Python versions. In particular, I do not have Python 3.4.0 or greater. Can I still use the Yocto Project?

You can get the required tools on your host development system a couple different ways (i.e. building a tarball or downloading a tarball). See the "[Required Git, tar, and Python Versions](#)" section for steps on how to update your build tools.

### 35.3. How can you claim Poky / OpenEmbedded-Core is stable?

There are three areas that help with stability;

- The Yocto Project team keeps [OE-Core](#) small and focused, containing around 830 recipes as opposed to the thousands available in other OpenEmbedded community layers. Keeping it small makes it easy to test and maintain.
- The Yocto Project team runs manual and automated tests using a small, fixed set of reference hardware as well as emulated targets.
- The Yocto Project uses an autobuilder, which provides continuous build and integration tests.

### 35.4. How do I get support for my board added to the Yocto Project?

Support for an additional board is added by creating a Board Support Package (BSP) layer for it. For more information on how to create a BSP layer, see the "[Understanding and Creating Layers](#)" section in the Yocto Project Development Tasks Manual and the Yocto Project Board Support Package (BSP) Developer's Guide.

Usually, if the board is not completely exotic, adding support in the Yocto Project is fairly straightforward.

### 35.5. Are there any products built using the OpenEmbedded build system?

The software running on the [Vernier LabQuest](#) is built using the OpenEmbedded build system. See the [Vernier LabQuest](#) website for more information. There are a number of pre-production devices using the OpenEmbedded build system and the Yocto Project team announces them as soon as they are released.

### 35.6. What does the OpenEmbedded build system produce as output?

Because you can use the same set of recipes to create output of various formats, the output of an OpenEmbedded build depends on how you start it. Usually, the output is a flashable image ready for the target device.

### 35.7. How do I add my package to the Yocto Project?

To add a package, you need to create a BitBake recipe. For information on how to create a BitBake recipe, see the "[Writing a New Recipe](#)" section in the Yocto Project Development Tasks Manual.



---

**35.8.** Do I have to reflash my entire board with a new Yocto Project image when recompiling a package?

The OpenEmbedded build system can build packages in various formats such as IPK for OPKG, Debian package (.deb), or RPM. You can then upgrade the packages using the package tools on the device, much like on a desktop distribution such as Ubuntu or Fedora. However, package management on the target is entirely optional.

---

**35.9.** I see the error 'chmod: XXXXX new permissions are r-xrwxrwx, not r-xr-xr-x'. What is wrong?

You are probably running the build on an NTFS filesystem. Use `ext2`, `ext3`, or `ext4` instead.

---

**35.10.** I see lots of 404 responses for files when the OpenEmbedded build system is trying to download sources. Is something wrong?

Nothing is wrong. The OpenEmbedded build system checks any configured source mirrors before downloading from the upstream sources. The build system does this searching for both source archives and pre-checked out versions of SCM-managed software. These checks help in large installations because it can reduce load on the SCM servers themselves. The address above is one of the default mirrors configured into the build system. Consequently, if an upstream source disappears, the team can place sources there so builds continue to work.

---

**35.11.** I have machine-specific data in a package for one machine only but the package is being marked as machine-specific in all cases, how do I prevent this?

Set `SRC_URI_OVERRIDES_PACKAGE_ARCH = "0"` in the `.bb` file but make sure the package is manually marked as machine-specific for the case that needs it. The code that handles `SRC_URI_OVERRIDES_PACKAGE_ARCH` is in the `meta/classes/base.bbclass` file.

---

**35.12.** I'm behind a firewall and need to use a proxy server. How do I do that?

Most source fetching by the OpenEmbedded build system is done by `wget` and you therefore need to specify the proxy settings in a `.wgetrc` file, which can be in your home directory if you are a single user or can be in `/usr/local/etc/wgetrc` as a global user file.

Following is the applicable code for setting various proxy types in the `.wgetrc` file. By default, these settings are disabled with comments. To use them, remove the comments:

```
You can set the default proxies for Wget to use for http, https, and ftp.
They will override the value in the environment.
#https_proxy = http://proxy.yoyodyne.com:18023/
#http_proxy = http://proxy.yoyodyne.com:18023/
#ftp_proxy = http://proxy.yoyodyne.com:18023/

If you do not want to use proxy at all, set this to off.
#use_proxy = on
```

The Yocto Project also includes a `meta-poky/conf/site.conf.sample` file that shows how to configure CVS and Git proxy servers if needed. For more information on setting up various proxy types and configuring proxy servers, see the "[Working Behind a Network Proxy](#)" Wiki page.

---

**35.13.** What's the difference between `target` and `target-native`?

The `*-native` targets are designed to run on the system being used for the build. These are usually tools that are needed to assist the build in some way such as `quilt-native`, which is used to apply patches. The non-native version is the one that runs on the target device.

---

**35.14.** I'm seeing random build failures. Help?!

If the same build is failing in totally different and random ways, the most likely explanation is:

- The hardware you are running the build on has some problem.
- You are running the build under virtualization, in which case the virtualization probably has bugs.

The OpenEmbedded build system processes a massive amount of data that causes lots of network, disk and CPU activity and is sensitive to even single-bit failures in any of these areas. True random failures have always been traced back to hardware or virtualization issues.

---

**35.15.** When I try to build a native recipe, the build fails with `iconv.h` problems.

If you get an error message that indicates GNU `libiconv` is not in use but `iconv.h` has been included from `libiconv`, you need to check to see if you have a previously installed version of the header file in `/usr/local/include`.

```
#error GNU libiconv not in use but included iconv.h is from libiconv
```

If you find a previously installed file, you should either uninstall it or temporarily rename it and try the build again.

This issue is just a single manifestation of "system leakage" issues caused when the OpenEmbedded build system finds and uses previously installed files during a native build. This type of issue might not be limited to `iconv.h`. Be sure that leakage cannot occur from `/usr/local/include` and `/opt` locations.

---

### 35.16. What do we need to ship for license compliance?

This is a difficult question and you need to consult your lawyer for the answer for your specific case. It is worth bearing in mind that for GPL compliance, there needs to be enough information shipped to allow someone else to rebuild and produce the same end result you are shipping. This means sharing the source code, any patches applied to it, and also any configuration information about how that package was configured and built.

You can find more information on licensing in the "[Licensing](#)" section in the Yocto Project Overview and Concepts Manual and also in the "[Maintaining Open Source License Compliance During Your Product's Lifecycle](#)" section in the Yocto Project Development Tasks Manual.

---

### 35.17. How do I disable the cursor on my touchscreen device?

You need to create a form factor file as described in the "[Miscellaneous BSP-Specific Recipe Files](#)" section in the Yocto Project Board Support Packages (BSP) Developer's Guide. Set the `HAVE_TOUCHSCREEN` variable equal to one as follows:

```
HAVE_TOUCHSCREEN=1
```

---

### 35.18. How do I make sure connected network interfaces are brought up by default?

The default interfaces file provided by the netbase recipe does not automatically bring up network interfaces. Therefore, you will need to add a BSP-specific netbase that includes an interfaces file. See the "[Miscellaneous BSP-Specific Recipe Files](#)" section in the Yocto Project Board Support Packages (BSP) Developer's Guide for information on creating these types of miscellaneous recipe files.

For example, add the following files to your layer:

```
meta-MACHINE/recipes-bsp/netbase/netbase/MACHINE/interfaces
meta-MACHINE/recipes-bsp/netbase/netbase_5.0.bbappend
```

---

### 35.19. How do I create images with more free space?

By default, the OpenEmbedded build system creates images that are 1.3 times the size of the populated root filesystem. To affect the image size, you need to set various configurations:

- **Image Size:** The OpenEmbedded build system uses the `IMAGE_ROOTFS_SIZE` variable to define the size of the image in Kbytes. The build system determines the size by taking into account the initial root filesystem size before any modifications such as requested size for the image and any requested additional free disk space to be added to the image.
- **Overhead:** Use the `IMAGE_OVERHEAD_FACTOR` variable to define the multiplier that the build system applies to the initial image size, which is 1.3 by default.
- **Additional Free Space:** Use the `IMAGE_ROOTFS_EXTRA_SPACE` variable to add additional free space to the image. The build system adds this space to the image after it determines its `IMAGE_ROOTFS_SIZE`.

---

### 35.20. Why don't you support directories with spaces in the pathnames?

The Yocto Project team has tried to do this before but too many of the tools the OpenEmbedded build system depends on, such as `autoconf`, break when they find spaces in pathnames. Until that situation changes, the team will not support spaces in pathnames.

---

### 35.21. How do I use an external toolchain?

The toolchain configuration is very flexible and customizable. It is primarily controlled with the `TCMODE` variable. This variable controls which `tcmode-*.inc` file to include from the `meta/conf/distro/include` directory within the [Source Directory](#).

The default value of `TCMODE` is "default", which tells the OpenEmbedded build system to use its internally built toolchain (i.e. `tcmode-default.inc`). However, other patterns are accepted. In particular, "external-\*" refers to external toolchains. One example is the Sourcery G++ Toolchain. The support for this toolchain resides in the separate `meta-sourcery` layer at <http://github.com/MentorEmbedded/meta-sourcery/>.

In addition to the toolchain configuration, you also need a corresponding toolchain recipe file. This recipe file needs to package up any pre-built objects in the toolchain such as `libgcc`, `libstdc++`, any locales, and `libc`.

---

### 35.22. How does the OpenEmbedded build system obtain source code and will it work behind my firewall or proxy server?

The way the build system obtains source code is highly configurable. You can setup the build system to get source code in most environments if HTTP transport is available.

When the build system searches for source code, it first tries the local download directory. If that location fails, Poky tries [PREMIRRORS](#), the upstream source, and then [MIRRORS](#) in that order.

Assuming your distribution is "poky", the OpenEmbedded build system uses the Yocto Project source `PREMIRRORS` by default for SCM-based sources, upstreams for normal tarballs, and then falls back to a number of other mirrors including the

Yocto Project source mirror if those fail.

As an example, you could add a specific server for the build system to attempt before any others by adding something like the following to the `local.conf` configuration file:

```
PREMIRRORS_prepend = "\
git://.*.* http://www.yoctoproject.org/sources/ \n \
ftp://.*.* http://www.yoctoproject.org/sources/ \n \
http://.*.* http://www.yoctoproject.org/sources/ \n \
https://.*.* http://www.yoctoproject.org/sources/ \n"
```

These changes cause the build system to intercept Git, FTP, HTTP, and HTTPS requests and direct them to the `http://sources.mirror`. You can use `file://` URLs to point to local directories or network shares as well.

Aside from the previous technique, these options also exist:

```
BB_NO_NETWORK = "1"
```

This statement tells BitBake to issue an error instead of trying to access the Internet. This technique is useful if you want to ensure code builds only from local sources.

Here is another technique:

```
BB_FETCH_PREMIRRORONLY = "1"
```

This statement limits the build system to pulling source from the `PREMIRRORS` only. Again, this technique is useful for reproducing builds.

Here is another technique:

```
BB_GENERATE_MIRROR_TARBALLS = "1"
```

This statement tells the build system to generate mirror tarballs. This technique is useful if you want to create a mirror server. If not, however, the technique can simply waste time during the build.

Finally, consider an example where you are behind an HTTP-only firewall. You could make the following changes to the `local.conf` configuration file as long as the `PREMIRRORS` server is current:

```
PREMIRRORS_prepend = "\
ftp://.*.* http://www.yoctoproject.org/sources/ \n \
http://.*.* http://www.yoctoproject.org/sources/ \n \
https://.*.* http://www.yoctoproject.org/sources/ \n"
BB_FETCH_PREMIRRORONLY = "1"
```

These changes would cause the build system to successfully fetch source over HTTP and any network accesses to anything other than the `PREMIRRORS` would fail.

The build system also honors the standard shell environment variables `http_proxy`, `ftp_proxy`, `https_proxy`, and `all_proxy` to redirect requests through proxy servers.

## Note

You can find more information on the "[Working Behind a Network Proxy](#)" Wiki page.

---

### 35.23. Can I get rid of build output so I can start over?

Yes - you can easily do this. When you use BitBake to build an image, all the build output goes into the directory created when you run the build environment setup script (i.e. `oe-init-build-env`). By default, this [Build Directory](#) is named `build` but can be named anything you want.

Within the Build Directory, is the `tmp` directory. To remove all the build output yet preserve any source code or downloaded files from previous builds, simply remove the `tmp` directory.

---

### 35.24. Why do `${bindir}` and `${libdir}` have strange values for `-native` recipes?

Executables and libraries might need to be used from a directory other than the directory into which they were initially installed. Complicating this situation is the fact that sometimes these executables and libraries are compiled with the expectation of being run from that initial installation target directory. If this is the case, moving them causes problems.

This scenario is a fundamental problem for package maintainers of mainstream Linux distributions as well as for the OpenEmbedded build system. As such, a well-established solution exists. Makefiles, Autotools configuration scripts, and other build systems are expected to respect environment variables such as `bindir`, `libdir`, and `sysconfdir` that indicate where executables, libraries, and data reside when a program is actually run. They are also expected to respect a `DESTDIR` environment variable, which is prepended to all the other variables when the build system actually installs the files. It is understood that the program does not actually run from within `DESTDIR`.

When the OpenEmbedded build system uses a recipe to build a target-architecture program (i.e. one that is intended for inclusion on the image being built), that program eventually runs from the root file system of that image. Thus, the build system provides a value of `"/usr/bin"` for `bindir`, a value of `"/usr/lib"` for `libdir`, and so forth.

Meanwhile, `DESTDIR` is a path within the [Build Directory](#). However, when the recipe builds a native program (i.e. one that is intended to run on the build machine), that program is never installed directly to the build machine's root file system. Consequently, the build system uses paths within the Build Directory for `DESTDIR`, `bindir` and related variables. To better understand this, consider the following two paths where the first is relatively normal and the second is not:

## Note

Due to these lengthy examples, the paths are artificially broken across lines for readability.

```

/home/maxtothemax/poky-bootchart2/build/tmp/work/i586-poky-linux/zlib/
1.2.8-r0/sysroot-destdir/usr/bin

/home/maxtothemax/poky-bootchart2/build/tmp/work/x86_64-linux/
zlib-native/1.2.8-r0/sysroot-destdir/home/maxtothemax/poky-bootchart2/
build/tmp/sysroots/x86_64-linux/usr/bin

```

Even if the paths look unusual, they both are correct - the first for a target and the second for a native recipe. These paths are a consequence of the DESTDIR mechanism and while they appear strange, they are correct and in practice very effective.

**35.25.** The files provided by my `*-native` recipe do not appear to be available to other recipes. Files are missing from the native sysroot, my recipe is installing to the wrong place, or I am getting permissions errors during the `do_install` task in my recipe! What is wrong?

This situation results when a build system does not recognize the environment variables supplied to it by [BitBake](#). The incident that prompted this FAQ entry involved a Makefile that used an environment variable named `BINDIR` instead of the more standard variable `bindir`. The makefile's hardcoded default value of `"/usr/bin"` worked most of the time, but not for the recipe's `-native` variant. For another example, permissions errors might be caused by a Makefile that ignores `DESTDIR` or uses a different name for that environment variable. Check the the build system to see if these kinds of issues exist.

## Chapter 36. Contributions and Additional Information¶

[36.1. Introduction](#)  
[36.2. Contributions](#)  
[36.3. Yocto Project Bugzilla](#)  
[36.4. Mailing lists](#)  
[36.5. Internet Relay Chat \(IRC\)](#)  
[36.6. Links and Related Documentation](#)

### 36.1. Introduction¶

The Yocto Project team is happy for people to experiment with the Yocto Project. A number of places exist to find help if you run into difficulties or find bugs. This presents information about contributing and participating in the Yocto Project.

### 36.2. Contributions¶

The Yocto Project gladly accepts contributions. You can submit changes to the project either by creating and sending pull requests, or by submitting patches through email. For information on how to do both as well as information on how to identify the maintainer for each area of code, see the ["Submitting a Change to the Yocto Project"](#) section in the Yocto Project Development Tasks Manual.

### 36.3. Yocto Project Bugzilla¶

The Yocto Project uses its own implementation of [Bugzilla](#) to track defects (bugs). Implementations of Bugzilla work well for group development because they track bugs and code changes, can be used to communicate changes and problems with developers, can be used to submit and review patches, and can be used to manage quality assurance.

Sometimes it is helpful to submit, investigate, or track a bug against the Yocto Project itself (e.g. when discovering an issue with some component of the build system that acts contrary to the documentation or your expectations).

A general procedure and guidelines exist for when you use Bugzilla to submit a bug. For information on how to use Bugzilla to submit a bug against the Yocto Project, see the following:

- The ["Submitting a Defect Against the Yocto Project"](#) section in the Yocto Project Development Tasks Manual.
- The Yocto Project [Bugzilla wiki page](#)

For information on Bugzilla in general, see <http://www.bugzilla.org/about/>.

### 36.4. Mailing lists¶

A number of mailing lists maintained by the Yocto Project exist as well as related OpenEmbedded mailing lists for discussion, patch submission and announcements. To subscribe to one of the following mailing lists, click on the appropriate URL in the following list and follow the instructions:

- <http://lists.yoctoproject.org/listinfo/yocto> - General Yocto Project discussion mailing list.
- <http://lists.openembedded.org/mailman/listinfo/openembedded-core> - Discussion mailing list about OpenEmbedded-Core (the core metadata).

- <http://lists.openembedded.org/mailman/listinfo/openembedded-devel> - Discussion mailing list about OpenEmbedded.
- <http://lists.openembedded.org/mailman/listinfo/bitbake-devel> - Discussion mailing list about the [BitBake](#) build tool.
- <http://lists.yoctoproject.org/listinfo/poky> - Discussion mailing list about Poky.
- <http://lists.yoctoproject.org/listinfo/yocto-announce> - Mailing list to receive official Yocto Project release and milestone announcements.

For more Yocto Project-related mailing lists, see the [Yocto Project Website](#).

### 36.5. Internet Relay Chat (IRC)

Two IRC channels on freenode are available for the Yocto Project and Poky discussions:

- #yocto
- #poky

### 36.6. Links and Related Documentation

Here is a list of resources you might find helpful:

- **The Yocto Project website:** The home site for the Yocto Project.
- **The Yocto Project Main Wiki Page:** The main wiki page for the Yocto Project. This page contains information about project planning, release engineering, QA & automation, a reference site map, and other resources related to the Yocto Project.
- **OpenEmbedded:** The build system used by the Yocto Project. This project is the upstream, generic, embedded distribution from which the Yocto Project derives its build system (Poky) and to which it contributes.
- **BitBake:** The tool used to process metadata.
- **BitBake User Manual:** A comprehensive guide to the BitBake tool. If you want information on BitBake, see this manual.
- **Yocto Project Quick Build:** This short document lets you experience building an image using the Yocto Project without having to understand any concepts or details.
- **Yocto project Overview and Concepts Manual:** This manual provides overview and conceptual information about the Yocto Project.
- **Yocto Project Development Tasks Manual:** This manual is a "how-to" guide that presents procedures useful to both application and system developers who use the Yocto Project.
- **Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual:** This guide provides information that lets you get going with the standard or extensible SDK. An SDK, with its cross-development toolchains, allows you to develop projects inside or outside of the Yocto Project environment.
- **Yocto Project Board Support Package (BSP) Developer's Guide:** This guide defines the structure for BSP components. Having a commonly understood structure encourages standardization.
- **Yocto Project Linux Kernel Development Manual:** This manual describes how to work with Linux Yocto kernels as well as provides a bit of conceptual information on the construction of the Yocto Linux kernel tree.
- **Yocto Project Reference Manual:** This manual provides reference material such as variable, task, and class descriptions.
- **Yocto Project Mega-Manual:** This manual is simply a single HTML file comprised of the bulk of the Yocto Project manuals. The Mega-Manual primarily exists as a vehicle by which you can easily search for phrases and terms used in the Yocto Project documentation set.
- **Yocto Project Profiling and Tracing Manual:** This manual presents a set of common and generally useful tracing and profiling schemes along with their applications (as appropriate) to each tool.
- **Toaster User Manual:** This manual introduces and describes how to set up and use Toaster. Toaster is an Application Programming Interface (API) and web-based interface to the [OpenEmbedded Build System](#), which uses BitBake, that reports build information.
- **FAQ:** A list of commonly asked questions and their answers.
- **Release Notes:** Features, updates and known issues for the current release of the Yocto Project. To access the Release Notes, go to the [Downloads](#) page on the Yocto Project website and click on the "RELEASE INFORMATION" link for the appropriate release.
- **Bugzilla:** The bug tracking application the Yocto Project uses. If you find problems with the Yocto Project, you should report them using this application.
- **Bugzilla Configuration and Bug Tracking Wiki Page:** Information on how to get set up and use the Yocto Project implementation of Bugzilla for logging and tracking Yocto Project defects.
- **Internet Relay Chat (IRC):** Two IRC channels on freenode are available for Yocto Project and Poky discussions: #yocto and #poky, respectively.
- **Quick EMUlator (QEMU):** An open-source machine emulator and virtualizer.



# Toaster User Manual

## Chapter 37. Introduction¶

### 37.1. Toaster Features¶

### 37.2. Installation Options¶

Toaster is a web interface to the Yocto Project's [OpenEmbedded build system](#). The interface enables you to configure and run your builds. Information about builds is collected and stored in a database. You can use Toaster to configure and start builds on multiple remote build servers.

### 37.1. Toaster Features¶

Toaster allows you to configure and run builds, and it provides extensive information about the build process.

- **Configure and Run Builds:** You can use the Toaster web interface to configure and start your builds. Builds started using the Toaster web interface are organized into projects. When you create a project, you are asked to select a release, or version of the build system you want to use for the project builds. As shipped, Toaster supports Yocto Project releases 1.8 and beyond. With the Toaster web interface, you can:
  - Browse layers listed in the various [layer sources](#) that are available in your project (e.g. the OpenEmbedded Layer Index at <http://layers.openembedded.org/layerindex/>).
  - Browse images, recipes, and machines provided by those layers.
  - Import your own layers for building.
  - Add and remove layers from your configuration.
  - Set configuration variables.
  - Select a target or multiple targets to build.
  - Start your builds.

Toaster also allows you to configure and run your builds from the command line, and switch between the command line and the web interface at any time. Builds started from the command line appear within a special Toaster project called "Command line builds".

- **Information About the Build Process:** Toaster also records extensive information about your builds. Toaster collects data for builds you start from the web interface and from the command line as long as Toaster is running.

### Note

You must start Toaster before the build or it will not collect build data.

With Toaster you can:

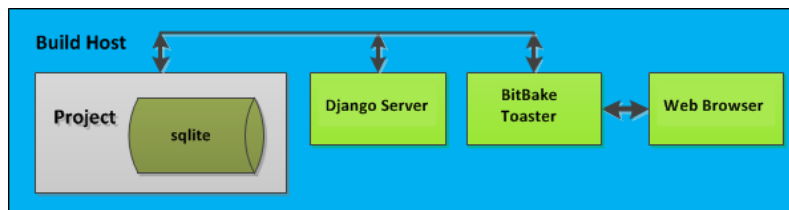
- See what was built (recipes and packages) and what packages were installed into your final image.
- Browse the directory structure of your image.
- See the value of all variables in your build configuration, and which files set each value.
- Examine error, warning, and trace messages to aid in debugging.
- See information about the BitBake tasks executed and reused during your build, including those that used shared state.
- See dependency relationships between recipes, packages, and tasks.
- See performance information such as build time, task time, CPU usage, and disk I/O.

For an overview of Toaster shipped with the Yocto Project 3.0.1 Release, see the "[Toaster - Yocto Project 2.2](#)" video.

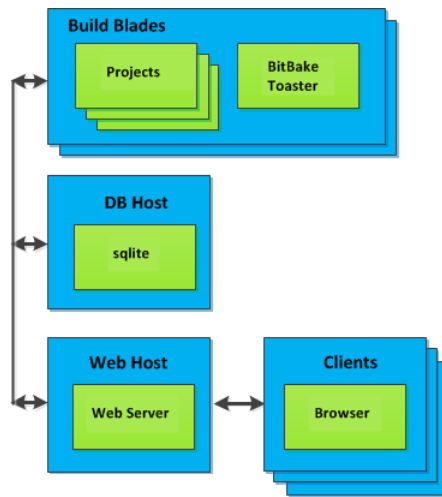
### 37.2. Installation Options¶

You can set Toaster up to run as a local instance or as a shared hosted service.

When Toaster is set up as a local instance, all the components reside on a single build host. Fundamentally, a local instance of Toaster is suited for a single user developing on a single build host.



Toaster as a hosted service is suited for multiple users developing across several build hosts. When Toaster is set up as a hosted service, its components can be spread across several machines:



## Chapter 38. Preparing to Use Toaster¶

[38.1. Setting Up the Basic System Requirements](#)

[38.2. Establishing Toaster System Dependencies](#)

[38.2.1. Install Toaster Packages](#)

This chapter describes how you need to prepare your system in order to use Toaster.

### 38.1. Setting Up the Basic System Requirements¶

Before you can use Toaster, you need to first set up your build system to run the Yocto Project. To do this, follow the instructions in the "Preparing the Build Host" section of the Yocto Project Development Tasks Manual. For Ubuntu/Debian, you might also need to do an additional install of pip3.

```
$ sudo apt-get install python3-pip
```

### 38.2. Establishing Toaster System Dependencies¶

Toaster requires extra Python dependencies in order to run. A Toaster requirements file named `toaster-requirements.txt` defines the Python dependencies. The requirements file is located in the `bitbake` directory, which is located in the root directory of the [Source Directory](#) (e.g. `poky/bitbake/toaster-requirements.txt`). The dependencies appear in a `pip`, install-compatible format.

#### 38.2.1. Install Toaster Packages¶

You need to install the packages that Toaster requires. Use this command:

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

The previous command installs the necessary Toaster modules into a local python 3 cache in your `$HOME` directory. The caches is actually located in `$HOME/.local`. To see what packages have been installed into your `$HOME` directory, do the following:

```
$ pip3 list installed --local
```

If you need to remove something, the following works:

```
$ pip3 uninstall PackageNameToUninstall
```

## Chapter 39. Setting Up and Using Toaster¶

[39.1. Starting Toaster for Local Development](#)

[39.2. Setting a Different Port](#)

[39.3. Setting Up Toaster Without a Web Server](#)

[39.4. Setting Up Toaster Without a Build Server](#)

[39.5. Setting up External Access](#)

[39.6. The Directory for Cloning Layers](#)

[39.7. The Build Directory](#)

[39.8. Creating a Django Superuser](#)

[39.9. Setting Up a Production Instance of Toaster](#)

[39.9.1. Requirements](#)

[39.9.2. Installation](#)

[39.10. Using the Toaster Web Interface](#)

[39.10.1. Toaster Web Interface Videos](#)

[39.10.2. Additional Information About the Local Yocto Project Release](#)

[39.10.3. Building a Specific Recipe Given Multiple Versions](#)

### 39.1. Starting Toaster for Local Development¶

Once you have set up the Yocto Project and installed the Toaster system dependencies as described in the "[Preparing to Use Toaster](#)" chapter, you are ready to start Toaster.

Navigate to the root of your [Source Directory](#) (e.g. poky):

```
$ cd poky
```

Once in that directory, source the build environment script:

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

Next, from the build directory (e.g. poky/build), start Toaster using this command:

```
$ source toaster start
```

You can now run your builds from the command line, or with Toaster as explained in section "[Using the Toaster Web Interface](#)".

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

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

### 39.2. Setting a Different Port¶

By default, Toaster starts on port 8000. You can use the `WEBPORT` parameter to set a different port. For example, the following command sets the port to "8400":

```
$ source toaster start webport=8400
```

### 39.3. Setting Up Toaster Without a Web Server¶

You can start a Toaster environment without starting its web server. This is useful for the following:

- Capturing a command-line build's statistics into the Toaster database for examination later.
- Capturing a command-line build's statistics when the Toaster server is already running.
- Having one instance of the Toaster web server track and capture multiple command-line builds, where each build is started in its own "noweb" Toaster environment.

The following commands show how to start a Toaster environment without starting its web server, perform BitBake operations, and then shut down the Toaster environment. Once the build is complete, you can close the Toaster environment. Before closing the environment, however, you should allow a few minutes to ensure the complete transfer of its BitBake build statistics to the Toaster database. If you have a separate Toaster web server instance running, you can watch this command-line build's progress and examine the results as soon as they are posted:

```
$ source toaster start noweb
$ bitbake target
$ source toaster stop
```

### 39.4. Setting Up Toaster Without a Build Server¶

You can start a Toaster environment with the "New Projects" feature disabled. Doing so is useful for the following:

- Sharing your build results over the web server while blocking others from starting builds on your host.
- Allowing only local command-line builds to be captured into the Toaster database.

Use the following command to set up Toaster without a build server:

```
$ source toaster start nobuild webport=port
```

### 39.5. Setting up External Access¶

By default, Toaster binds to the loop back address (i.e. localhost), which does not allow access from external hosts. To allow external access, use the `WEBPORT` parameter to open an address that connects to the network, specifically the IP address that your NIC uses to connect to the network. You can also bind to all IP addresses the computer supports by using the shortcut "0.0.0.0:port".

The following example binds to all IP addresses on the host:

```
$ source toaster start webport=0.0.0.0:8400
```

This example binds to a specific IP address on the host's NIC:

```
$ source toaster start webport=192.168.1.1:8400
```

### 39.6. The Directory for Cloning Layers¶



Toaster creates a `_toaster_clones` directory inside your Source Directory (i.e. `poky`) to clone any layers needed for your builds.

Alternatively, if you would like all of your Toaster related files and directories to be in a particular location other than the default, you can set the `TOASTER_DIR` environment variable, which takes precedence over your current working directory. Setting this environment variable causes Toaster to create and use `$TOASTER_DIR/_toaster_clones`.

### 39.7. The Build Directory¶

Toaster creates a build directory within your Source Directory (e.g. `poky`) to execute the builds.

Alternatively, if you would like all of your Toaster related files and directories to be in a particular location, you can set the `TOASTER_DIR` environment variable, which takes precedence over your current working directory. Setting this environment variable causes Toaster to use `$TOASTER_DIR/build` as the build directory.

### 39.8. Creating a Django Superuser¶

Toaster is built on the [Django framework](#). Django provides an administration interface you can use to edit Toaster configuration parameters.

To access the Django administration interface, you must create a superuser by following these steps:

1. If you used `pip3`, which is recommended, to set up the Toaster system dependencies, you need be sure the local user path is in your `PATH` list. To append the `pip3` local user path, use the following command:

```
$ export PATH=$PATH:$HOME/.local/bin
```

2. From the directory containing the Toaster database, which by default is the [Build Directory](#), invoke the `createsuperuser` command from `manage.py`:

```
$ cd ~/poky/build
$../bitbake/lib/toaster/manage.py createsuperuser
```

3. Django prompts you for the username, which you need to provide.
4. Django prompts you for an email address, which is optional.
5. Django prompts you for a password, which you must provide.
6. Django prompts you to re-enter your password for verification.

After completing these steps, the following confirmation message appears:

```
Superuser created successfully.
```

Creating a superuser allows you to access the Django administration interface through a browser. The URL for this interface is the same as the URL used for the Toaster instance with `/admin` on the end. For example, if you are running Toaster locally, use the following URL:

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

You can use the Django administration interface to set Toaster configuration parameters such as the build directory, layer sources, default variable values, and BitBake versions.

### 39.9. Setting Up a Production Instance of Toaster¶

You can use a production instance of Toaster to share the Toaster instance with remote users, multiple users, or both. The production instance is also the setup that can handle heavier loads on the web service. Use the instructions in the following sections to set up Toaster to run builds through the Toaster web interface.

#### 39.9.1. Requirements¶

Be sure you meet the following requirements:

#### Note

You must comply with all Apache, `mod-wsgi`, and Mysql requirements.

- Have all the build requirements as described in the ["Preparing to Use Toaster"](#) chapter.
- Have an Apache webserver.
- Have `mod-wsgi` for the Apache webserver.
- Use the Mysql database server.
- If you are using Ubuntu 16.04, run the following:

```
$ sudo apt-get install apache2 libapache2-mod-wsgi-py3 mysql-server python3-pip libmysqlclient-dev
```

- If you are using Fedora 24 or a RedHat distribution, run the following:

```
$ sudo dnf install httpd python3-mod_wsgi python3-pip mariadb-server mariadb-devel python3-devel
```

- If you are using openSUSE Leap 42.1, run the following:

```
$ sudo zypper install apache2 apache2-mod_wsgi-python3 python3-pip mariadb mariadb-client python3-devel
```

### 39.9.2. Installation¶

Perform the following steps to install Toaster:

1. Create toaster user and set its home directory to `/var/www/toaster`:

```
$ sudo /usr/sbin/useradd toaster -md /var/www/toaster -s /bin/false
$ sudo su - toaster -s /bin/bash
```

2. Checkout a copy of poky into the web server directory. You will be using `/var/www/toaster`:

```
$ git clone git://git.yoctoproject.org/poky
$ git checkout zeus
```

3. Install Toaster dependencies using the `--user` flag which keeps the Python packages isolated from your system-provided packages:

```
$ cd /var/www/toaster/
$ pip3 install --user -r ./poky/bitbake/toaster-requirements.txt
$ pip3 install --user mysqlclient
```

#### Note

Isolating these packages is not required but is recommended. Alternatively, you can use your operating system's package manager to install the packages.

4. Configure Toaster by editing `/var/www/toaster/poky/bitbake/lib/toaster/toastermain/settings.py` as follows:

- Edit the DATABASES settings:

```
DATABASES = {
 'default': {
 'ENGINE': 'django.db.backends.mysql',
 'NAME': 'toaster_data',
 'USER': 'toaster',
 'PASSWORD': 'yourpasswordhere',
 'HOST': 'localhost',
 'PORT': '3306',
 }
}
```

- Edit the SECRET\_KEY:

```
SECRET_KEY = 'your_secret_key'
```

- Edit the STATIC\_ROOT:

```
STATIC_ROOT = '/var/www/toaster/static_files/'
```

5. Add the database and user to the `mysql` server defined earlier:

```
$ mysql -u root -p
mysql> CREATE DATABASE toaster_data;
mysql> CREATE USER 'toaster'@'localhost' identified by 'yourpasswordhere';
mysql> GRANT all on toaster_data.* to 'toaster'@'localhost';
mysql> quit
```

6. Get Toaster to create the database schema, default data, and gather the statically-served files:

```
$ cd /var/www/toaster/poky/
$./bitbake/lib/toaster/manage.py migrate
$ TOASTER_DIR=`pwd` TEMPLATECONF='poky' \
 ./bitbake/lib/toaster/manage.py checksettings
$./bitbake/lib/toaster/manage.py collectstatic
```

In the previous example, from the `poky` directory, the `migrate` command ensures the database schema changes have propagated correctly (i.e. migrations). The next line sets the Toaster root directory `TOASTER_DIR` and the location of the Toaster configuration file `TOASTER_CONF`, which is relative to `TOASTER_DIR`. The `TEMPLATECONF` value reflects the contents of `poky/.templateconf`, and by default, should include the string "poky". For more information on the Toaster configuration file, see the "[Configuring Toaster](#)" section.

This line also runs the `checksettings` command, which configures the location of the Toaster Build Directory. The Toaster root directory `TOASTER_DIR` determines where the Toaster build directory is created on the file system. In the example above, `TOASTER_DIR` is set as follows:

```
/var/www/toaster/poky
```

This setting causes the Toaster build directory to be:

```
/var/www/toaster/poky/build
```

Finally, the `collectstatic` command is a Django framework command that collects all the statically served files into a designated directory to be served up by the Apache web server as defined by `STATIC_ROOT`.

7. Test and/or use the Mysql integration with Toaster's Django web server. At this point, you can start up the normal Toaster Django web server with the Toaster database in Mysql. You can use this web server to confirm that the database migration and data population from the Layer Index is complete.

To start the default Toaster Django web server with the Toaster database now in Mysql, use the standard start commands:

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

Additionally, if Django is sufficient for your requirements, you can use it for your release system and migrate later to Apache as your requirements change.

8. Add an Apache configuration file for Toaster to your Apache web server's configuration directory. If you are using Ubuntu or Debian, put the file here:

```
/etc/apache2/conf-available/toaster.conf
```

If you are using Fedora or RedHat, put it here:

```
/etc/httpd/conf.d/toaster.conf
```

If you are using OpenSUSE, put it here:

```
/etc/apache2/conf.d/toaster.conf
```

Following is a sample Apache configuration for Toaster you can follow:

```
Alias /static /var/www/toaster/static_files
<Directory /var/www/toaster/static_files>
 <IfModule mod_access_compat.c>
 Order allow,deny
 Allow from all
 </IfModule>
 <IfModule !mod_access_compat.c>
 Require all granted
 </IfModule>
</Directory>

<Directory /var/www/toaster/poky/bitbake/lib/toaster/toastermain>
 <Files "wsgi.py">
 Require all granted
 </Files>
</Directory>

WSGIDaemonProcess toaster_wsgi python-path=/var/www/toaster/poky/bitbake/lib/toaster:/var/www/toaster/.local/lib/python
WSGIScriptAlias / "/var/www/toaster/poky/bitbake/lib/toaster/toastermain/wsgi.py"
<Location />
 WSGIProcessGroup toaster_wsgi
</Location>
```

If you are using Ubuntu or Debian, you will need to enable the config and module for Apache:

```
$ sudo a2enmod wsgi
$ sudo a2enconf toaster
$ chmod +x bitbake/lib/toaster/toastermain/wsgi.py
```

Finally, restart Apache to make sure all new configuration is loaded. For Ubuntu, Debian, and OpenSUSE use:

```
$ sudo service apache2 restart
```

For Fedora and RedHat use:

```
$ sudo service httpd restart
```

9. Prepare the systemd service to run Toaster builds. Here is a sample configuration file for the service:

```
[Unit]
Description=Toaster runbuilds

[Service]
Type=forking
User=toaster
ExecStart=/usr/bin/screen -d -m -S runbuilds /var/www/toaster/poky/bitbake/lib/toaster/runbuilds-service.sh start
ExecStop=/usr/bin/screen -S runbuilds -X quit
WorkingDirectory=/var/www/toaster/poky

[Install]
WantedBy=multi-user.target
```

Prepare the `runbuilds-service.sh` script that you need to place in the `/var/www/toaster/poky/bitbake/lib/toaster/` directory by setting up executable permissions:

```
#!/bin/bash

#export http_proxy=http://proxy.host.com:8080
#export https_proxy=http://proxy.host.com:8080
```

```
#export GIT_PROXY_COMMAND=$HOME/bin/gitproxy

cd ~/poky/
source ./oe-init-build-env build
source ../bitbake/bin/toaster $1 noweb
["$1" == 'start'] && /bin/bash
```

10. Run the service:

```
service runbuilds start
```

Since the service is running in a detached screen session, you can attach to it using this command:

```
$ sudo su - toaster
$ screen -r5 runbuilds
```

You can detach from the service again using "Ctrl-a" followed by "d" key combination.

You can now open up a browser and start using Toaster.

## 39.10. Using the Toaster Web Interface¶

The Toaster web interface allows you to do the following:

- Browse published layers in the [OpenEmbedded Layer Index](#) that are available for your selected version of the build system.
- Import your own layers for building.
- Add and remove layers from your configuration.
- Set configuration variables.
- Select a target or multiple targets to build.
- Start your builds.
- See what was built (recipes and packages) and what packages were installed into your final image.
- Browse the directory structure of your image.
- See the value of all variables in your build configuration, and which files set each value.
- Examine error, warning and trace messages to aid in debugging.
- See information about the BitBake tasks executed and reused during your build, including those that used shared state.
- See dependency relationships between recipes, packages and tasks.
- See performance information such as build time, task time, CPU usage, and disk I/O.

### 39.10.1. Toaster Web Interface Videos¶

Following are several videos that show how to use the Toaster GUI:

- **Build Configuration:** This [video](#) overviews and demonstrates build configuration for Toaster.
- **Build Custom Layers:** This [video](#) shows you how to build custom layers that are used with Toaster.
- **Toaster Homepage and Table Controls:** This [video](#) goes over the Toaster entry page, and provides an overview of the data manipulation capabilities of Toaster, which include search, sorting and filtering by different criteria.
- **Build Dashboard:** This [video](#) shows you the build dashboard, a page providing an overview of the information available for a selected build.
- **Image Information:** This [video](#) walks through the information Toaster provides about images: packages installed and root file system.
- **Configuration:** This [video](#) provides Toaster build configuration information.
- **Tasks:** This [video](#) shows the information Toaster provides about the tasks run by the build system.
- **Recipes and Packages Built:** This [video](#) shows the information Toaster provides about recipes and packages built.
- **Performance Data:** This [video](#) shows the build performance data provided by Toaster.

### 39.10.2. Additional Information About the Local Yocto Project Release¶

This section only applies if you have set up Toaster for local development, as explained in the ["Starting Toaster for Local Development"](#) section.

When you create a project in Toaster, you will be asked to provide a name and to select a Yocto Project release. One of the release options you will find is called "Local Yocto Project".

yoctoPROJECT

Toaster

Manual

Create a new project

Project name (required)

Release

✓ Local Yocto Project

Yocto Project 2.0 Jethro

Yocto Project master

h the

version of the Yocto Project you

have cloned or downloaded to your

computer.

Create project

To create a project, you need to enter a project name

When you select the "Local Yocto Project" release, Toaster will run your builds using the local Yocto Project clone you have in your computer: the same clone you are using to run Toaster. Unless you manually update this clone, your builds will always use the same Git revision.

If you select any of the other release options, Toaster will fetch the tip of your selected release from the upstream [Yocto Project repository](#) every time you run a build. Fetching this tip effectively means that if your selected release is updated upstream, the Git revision you are using for your builds will change. If you are doing development locally, you might not want this change to happen. In that case, the "Local Yocto Project" release might be the right choice.

However, the "Local Yocto Project" release will not provide you with any compatible layers, other than the three core layers that come with the Yocto Project:

- [openembedded-core](#)
- [meta-poky](#)
- [meta-yocto-bsp](#)

yoctoPROJECT

Toaster

All builds

All projects

New project

Manual

My project

Configuration

Builds (0)

Import layer

New custom image

Type the recipe you want to build

Build

Configuration

COMPATIBLE METADATA

Custom images

Image recipes

Software recipes

Machines

Layers

EXTRA CONFIGURATION

BitBake variables

Compatible layers (3)

Search compatible layers

Search

Edit columns

Show rows: 25

| Layer             | Summary       | Git revision | Dependencies | Add   Remove |
|-------------------|---------------|--------------|--------------|--------------|
| meta-poky         |               | HEAD         |              | Remove layer |
| meta-yocto-bsp    |               | HEAD         |              | Remove layer |
| openembedded-core | Core metadata | HEAD         |              | Remove layer |

Show rows: 25

If you want to build any other layers, you will need to manually import them into your Toaster project, using the "Import layer" page.

yoctoPROJECT

Toaster

All builds

All projects

New project

Manual

My project

ConfigurationBuilds (0)Import layerNew custom image

Type the recipe you want to buildBuild

Layer repository information

The layer you are importing must be compatible with **Local Yocto Project**, which is the release you are using in this project.

Layer name

Git repository URL

Repository subdirectory (optional)

Git revision

Layer dependencies (optional)

openembedded-core

Type a layer nameAdd layer

You can only add layers Toaster knows about

Import and add to project

To import a layer you need to enter a layer name, a Git repository URL and a revision (branch, tag or commit)

39.10.3. Building a Specific Recipe Given Multiple Versions¶

Occasionally, a layer might provide more than one version of the same recipe. For example, the `openembedded-core` layer provides two versions of the `bash` recipe (i.e. 3.2.48 and 4.3.30-r0) and two versions of the `which` recipe (i.e. 2.21 and 2.18). The following figure shows this exact scenario:

yoctoPROJECT

Toaster

All builds

All projects

New build

New project

Manual

Demo project: Configuration → Compatible layers → openembedded-core (master)

openembedded-core (master)

Layer detailsRecipes (4)Machines (7)

Delete the openembedded-core layer from your project

bashSearchShow rows: 25

| Recipe | Description                                                                                                                                                                                                                                                  | Version   | Build recipe |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------|--------------|
| bash   | An sh-compatible command language interpreter                                                                                                                                                                                                                | 3.2.48    | Build recipe |
| bash   | An sh-compatible command language interpreter.                                                                                                                                                                                                               | 4.3.30-r0 | Build recipe |
| which  | Which is a utility that prints out the full path of the executables that <code>bash(1)</code> would execute when the passed program names would have been entered on the shell prompt. It does this by using the exact same algorithm as <code>bash</code> . | 2.21      | Build recipe |
| which  | Which is a utility that prints out the full path of the executables that <code>bash(1)</code> would execute when the passed program names would have been entered on the shell prompt. It does this by using the exact same algorithm as <code>bash</code> . | 2.18      | Build recipe |

Show rows: 25

About openembedded-core

Summary

Core metadata

Description

OpenEmbedded-Core is a layer containing the core metadata for current versions of OpenEmbedded. It is distro-less (can build a functional image with `DISTRO = ""`) and contains only emulated machine support.

By default, the OpenEmbedded build system builds one of the two recipes. For the `bash` case, version 4.3.30-r0 is built by default. Unfortunately, Toaster as it exists, is not able to override the default recipe version. If you would like to build `bash` 3.2.48, you need to

yocto

PROJECT

Toaster

All builds

All projects

New project

Manual

Demo project

Builds (1)

Configuration

Import layer

Type the recipe you want to build

Build

Configuration

COMPATIBLE METADATA

Image recipes

Software recipes

Machines

Layers

EXTRA CONFIGURATION

BitBake variables

Bitbake variables

DISTRO

poky

IMAGE\_FSTYPES

ext3 jffs2 tar.bz2

IMAGE\_INSTALL\_append

Not set

PACKAGE\_CLASSES

package\_rpm

SDKMACHINE

x86\_64

Add variable

Variable

Type variable name

Value

Type variable value

Add variable

Some variables are reserved from Toaster

Toaster cannot set any variables that impact 1) the configuration of the build servers, or 2) where artifacts produced by the build are stored. Such variables include:

BB\_DISKMON\_DIRS BB\_NUMBER\_THREADS CVS\_PROXY\_HOST CVS\_PROXY\_PORT DL\_DIR PARALLEL\_MAKE SSTATE\_DIR SSTATE\_MIRRORS TMPDIR

Plus the following standard shell environment variables:

http\_proxy ftp\_proxy https\_proxy all\_proxy

<https://www.yoctoproject.org/docs/current/mega-manual/mega-manual.html>

# Demo project

Builds (1)

Configuration

Import layer

Type the recipe you want to build

Build

Configuration

COMPATIBLE METADATA

Image recipes

Software recipes

Machines

Layers

EXTRA CONFIGURATION

BitBake variables

## Bitbake variables

- DISTRO** ⓘ  
poky 
- IMAGE\_FSTYPES** ⓘ  
ext3 jffs2 tar.bz2 
- IMAGE\_INSTALL\_append** ⓘ  
Not set 
- PACKAGE\_CLASSES** ⓘ  
package\_rpm 
- SDKMACHINE** ⓘ  
x86\_64 

### Add variable

Variable ⓘ

PREFERRED\_VERSION\_bash

Value

3.2.48

Add variable

**Some variables are reserved from Toaster**

Toaster cannot set any variables that impact 1) the configuration of the build servers, or 2) where artifacts produced by the build are stored. Such variables include:

BB\_DISKMON\_DIRS BB\_NUMBER\_THREADS CVS\_PROXY\_HOST  
CVS\_PROXY\_PORT DL\_DIR PARALLEL\_MAKE SSTATE\_DIR  
SSTATE\_MIRRORS TMPDIR

Plus the following standard shell environment variables:

http\_proxy ftp\_proxy https\_proxy all\_proxy

After clicking the "Add variable" button, the settings for PREFERRED\_VERSION are added to the bottom of the BitBake variables list. With these settings, the OpenEmbedded build system builds the desired version of the recipe rather than the default version:



# Demo project

Builds (1)

Configuration

Import layer

Type the recipe you want to build

Build

Configuration

COMPATIBLE METADATA

Recipes

Machines

Layers

EXTRA CONFIGURATION

BitBake variables

## Bitbake variables

- DISTROpoky
- IMAGE\_FSTYPESext3 jffs2 tar.bz2
- IMAGE\_INSTALL\_appendNot set
- PACKAGE\_CLASSESpackage\_rpm
- SDKMACHINEx86\_64
- PREFERRED\_VERSION\_bash3.2.48

### Add variable

Variable

Type variable name

Value

Type variable value

Add variable

Some variables are reserved from Toaster

Toaster cannot set any variables that impact 1) the configuration of the build servers, or 2) where artifacts produced by the build are stored. Such variables include:

BB\_DISKMON\_DIRS BB\_NUMBER\_THREADS CVS\_PROXY\_HOST  
CVS\_PROXY\_PORT DL\_DIR PARALLEL\_MAKE SSTATE\_DIR  
SSTATE\_MIRRORS TMPDIR

Plus the following standard shell environment variables:

http\_proxy ftp\_proxy https\_proxy all\_proxy

## Chapter 40. Concepts and Reference

- 40.1. Layer Source
- 40.1.1. Setting Up and Using a Layer Source
- 40.2. Releases
- 40.2.1. Pre-Configured Releases
- 40.3. Configuring Toaster
- 40.3.1. Configuring the Workflow
- 40.3.2. Customizing Pre-Set Data
- 40.3.3. Understanding Fixture File Format
- 40.4. Remote Toaster Monitoring
- 40.4.1. Checking Health
- 40.4.2. Determining Status of Builds in Progress
- 40.4.3. Checking Status of Builds Completed
- 40.4.4. Determining Status of a Specific Build
- 40.5. Useful Commands
- 40.5.1. buildsls
- 40.5.2. builddelete
- 40.5.3. perf
- 40.5.4. checksettings
- 40.5.5. runbuilds

In order to configure and use Toaster, you should understand some concepts and have some basic command reference material available. This final chapter provides conceptual information on layer sources, releases, and JSON configuration files. Also provided is a quick look at some useful `manage.py` commands that are Toaster-specific. Information on `manage.py` commands does exist across the Web and the information in this manual by no means attempts to provide a command comprehensive reference.

### 40.1. Layer Source

In general, a "layer source" is a source of information about existing layers. In particular, we are concerned with layers that you can use with the Yocto Project and Toaster. This chapter describes a particular type of layer source called a "layer index."

A layer index is a web application that contains information about a set of custom layers. A good example of an existing layer index is the OpenEmbedded Layer Index. A public instance of this layer index exists at <http://layers.openembedded.org>. You can find the code for this layer index's web application at <http://git.yoctoproject.org/cgit/cgit.cgi/layerindex-web/>.

When you tie a layer source into Toaster, it can query the layer source through a [REST API](#), store the information about the layers in the Toaster database, and then show the information to users. Users are then able to view that information and build layers from Toaster itself without worrying about cloning or editing the BitBake layers configuration file `bblayers.conf`.

Tying a layer source into Toaster is convenient when you have many custom layers that need to be built on a regular basis by a community of developers. In fact, Toaster comes pre-configured with the OpenEmbedded Metadata Index.

## Note

You do not have to use a layer source to use Toaster. Tying into a layer source is optional.

### 40.1.1. Setting Up and Using a Layer Source¶

To use your own layer source, you need to set up the layer source and then tie it into Toaster. This section describes how to tie into a layer index in a manner similar to the way Toaster ties into the OpenEmbedded Metadata Index.

#### 40.1.1.1. Understanding Your Layers¶

The obvious first step for using a layer index is to have several custom layers that developers build and access using the Yocto Project on a regular basis. This set of layers needs to exist and you need to be familiar with where they reside. You will need that information when you set up the code for the web application that "hooks" into your set of layers.

For general information on layers, see the ["The Yocto Project Layer Model"](#) section in the Yocto Project Overview and Concepts Manual. For information on how to create layers, see the ["Understanding and Creating Layers"](#) section in the Yocto Project Development Tasks Manual.

#### 40.1.1.2. Configuring Toaster to Hook Into Your Layer Index¶

If you want Toaster to use your layer index, you must host the web application in a server to which Toaster can connect. You also need to give Toaster the information about your layer index. In other words, you have to configure Toaster to use your layer index. This section describes two methods by which you can configure and use your layer index.

In the previous section, the code for the OpenEmbedded Metadata Index (i.e. <http://layers.openembedded.org>) was referenced. You can use this code, which is at <http://git.yoctoproject.org/cgit/cgit.cgi/layerindex-web/>, as a base to create your own layer index.

##### 40.1.1.2.1. Use the Administration Interface¶

Access the administration interface through a browser by entering the URL of your Toaster instance and adding `/admin` to the end of the URL. As an example, if you are running Toaster locally, use the following URL:

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

The administration interface has a "Layer sources" section that includes an "Add layer source" button. Click that button and provide the required information. Make sure you select "layerindex" as the layer source type.

##### 40.1.1.2.2. Use the Fixture Feature¶

The Django fixture feature overrides the default layer server when you use it to specify a custom URL. To use the fixture feature, create (or edit) the file `bitbake/lib/toaster.orm/fixtures/custom.xml`, and then set the following Toaster setting to your custom URL:

```
<?xml version="1.0" ?>
<django-objects version="1.0">
 <object model="orm.toastersetting" pk="100">
 <field name="name" type="CharField">CUSTOM_LAYERINDEX_SERVER</field>
 <field name="value" type="CharField">https://layers.my_organization.org/layerindex/branch/master/layers/</field>
 </object>
</django-objects>
```

When you start Toaster for the first time, or if you delete the file `toaster.sqlite` and restart, the database will populate cleanly from this layer index server.

Once the information has been updated, verify the new layer information is available by using the Toaster web interface. To do that, visit the "All compatible layers" page inside a Toaster project. The layers from your layer source should be listed there.

If you change the information in your layer index server, refresh the Toaster database by running the following command:

```
$ bitbake/lib/toaster/manage.py lsupdates
```

If Toaster can reach the API URL, you should see a message telling you that Toaster is updating the layer source information.

## 40.2. Releases¶

When you create a Toaster project using the web interface, you are asked to choose a "Release." In the context of Toaster, the term "Release" refers to a set of layers and a BitBake version the OpenEmbedded build system uses to build something. As shipped, Toaster is pre-configured with releases that correspond to Yocto Project release branches. However, you can modify, delete, and create new releases according to your needs. This section provides some background information on releases.

### 40.2.1. Pre-Configured Releases¶

As shipped, Toaster is configured to use a specific set of releases. Of course, you can always configure Toaster to use any release. For example, you might want your project to build against a specific commit of any of the "out-of-the-box" releases. Or, you might want your

project to build against different revisions of OpenEmbedded and BitBake.

As shipped, Toaster is configured to work with the following releases:

- **Yocto Project 3.0.1 "Zeus" or OpenEmbedded "Zeus"**: This release causes your Toaster projects to build against the head of the zeus branch at <http://git.yoctoproject.org/cgit/cgit.cgi/poky/log/?h=rocko> or <http://git.openembedded.org/openembedded-core/commit/?h=rocko>.
- **Yocto Project "Master" or OpenEmbedded "Master"**: This release causes your Toaster Projects to build against the head of the master branch, which is where active development takes place, at <http://git.yoctoproject.org/cgit/cgit.cgi/poky/log/> or <http://git.openembedded.org/openembedded-core/log/>.
- **Local Yocto Project or Local OpenEmbedded**: This release causes your Toaster Projects to build against the head of the poky or openembedded-core clone you have local to the machine running Toaster.

## 40.3. Configuring Toaster¶

In order to use Toaster, you must configure the database with the default content. The following subsections describe various aspects of Toaster configuration.

### 40.3.1. Configuring the Workflow¶

The `bldcontrol/management/commands/checksettings.py` file controls workflow configuration. The following steps outline the process to initially populate this database.

1. The default project settings are set from `orm/fixtures/settings.xml`.
2. The default project distro and layers are added from `orm/fixtures/poky.xml` if poky is installed. If poky is not installed, they are added from `orm/fixtures/oe-core.xml`.
3. If the `orm/fixtures/custom.xml` file exists, then its values are added.
4. The layer index is then scanned and added to the database.

Once these steps complete, Toaster is set up and ready to use.

### 40.3.2. Customizing Pre-Set Data¶

The pre-set data for Toaster is easily customizable. You can create the `orm/fixtures/custom.xml` file to customize the values that go into the database. Customization is additive, and can either extend or completely replace the existing values.

You use the `orm/fixtures/custom.xml` file to change the default project settings for the machine, distro, file images, and layers. When creating a new project, you can use the file to define the offered alternate project release selections. For example, you can add one or more additional selections that present custom layer sets or distros, and any other local or proprietary content.

Additionally, you can completely disable the content from the `oe-core.xml` and `poky.xml` files by defining the section shown below in the `settings.xml` file. For example, this option is particularly useful if your custom configuration defines fewer releases or layers than the default fixture files.

The following example sets "name" to "CUSTOM\_XML\_ONLY" and its value to "True".

```
<object model="orm.toastersetting" pk="99">
 <field type="CharField" name="name">CUSTOM_XML_ONLY</field>
 <field type="CharField" name="value">True</field>
</object>
```

### 40.3.3. Understanding Fixture File Format¶

The following is an overview of the file format used by the `oe-core.xml`, `poky.xml`, and `custom.xml` files.

The following subsections describe each of the sections in the fixture files, and outline an example section of the XML code. you can use to help understand this information and create a local `custom.xml` file.

#### 40.3.3.1. Defining the Default Distro and Other Values¶

This section defines the default distro value for new projects. By default, it reserves the first Toaster Setting record "1". The following demonstrates how to set the project default value for DISTRO:

```
<!-- Set the project default value for DISTRO -->
<object model="orm.toastersetting" pk="1">
 <field type="CharField" name="name">DEFCONF_DISTRO</field>
 <field type="CharField" name="value">poky</field>
</object>
```

You can override other default project values by adding additional Toaster Setting sections such as any of the settings coming from the `settings.xml` file. Also, you can add custom values that are included in the BitBake environment. The "pk" values must be unique. By convention, values that set default project values have a "DEFCONF" prefix.

#### 40.3.3.2. Defining BitBake Version¶

The following defines which version of BitBake is used for the following release selection:

```
<!-- Bitbake versions which correspond to the metadata release -->
<object model="orm.bitbakeversion" pk="1">
 <field type="CharField" name="name">rocko</field>
 <field type="CharField" name="giturl">git://git.yoctoproject.org/poky</field>
 <field type="CharField" name="branch">rocko</field>
 <field type="CharField" name="dirpath">bitbake</field>
```

```
</object>
```

#### 40.3.3.3. Defining Release¶

The following defines the releases when you create a new project.

```
<!-- Releases available -->
<object model="orm.release" pk="1">
 <field type="CharField" name="name">rocko</field>
 <field type="CharField" name="description">Yocto Project 2.4 "Rocko"</field>
 <field rel="ManyToOneRel" to="orm.bitbakeversion" name="bitbake_version">1</field>
 <field type="CharField" name="branch_name">rocko</field>
 <field type="TextField" name="helptext">Toaster will run your builds using the tip of the
</object>
```

The "pk" value must match the above respective BitBake version record.

#### 40.3.3.4. Defining the Release Default Layer Names¶

The following defines the default layers for each release:

```
<!-- Default project layers for each release -->
<object model="orm.releasedefaultlayer" pk="1">
 <field rel="ManyToOneRel" to="orm.release" name="release">1</field>
 <field type="CharField" name="layer_name">openembedded-core</field>
</object>
```

The 'pk' values in the example above should start at "1" and increment uniquely. You can use the same layer name in multiple releases.

#### 40.3.3.5. Defining Layer Definitions¶

Layer definitions are the most complex. The following defines each of the layers, and then defines the exact layer version of the layer used for each respective release. You must have one `orm.layer` entry for each layer. Then, with each entry you need a set of `orm.layer_version` entries that connects the layer with each release that includes the layer. In general all releases include the layer.

```
<object model="orm.layer" pk="1">
 <field type="CharField" name="name">openembedded-core</field>
 <field type="CharField" name="layer_index_url"></field>
 <field type="CharField" name="vcs_url">git://git.yoctoproject.org/poky</field>
 <field type="CharField" name="vcs_web_url">http://git.yoctoproject.org/cgi/cgit.cgi/poky</field>
 <field type="CharField" name="vcs_web_tree_base_url">http://git.yoctoproject.org/cgi/cgit.cgi/poky/tree/%path%?h=branch</field>
 <field type="CharField" name="vcs_web_file_base_url">http://git.yoctoproject.org/cgi/cgit.cgi/poky/tree/%path%?h=branch</field>
</object>
<object model="orm.layer_version" pk="1">
 <field rel="ManyToOneRel" to="orm.layer" name="layer">1</field>
 <field type="IntegerField" name="layer_source">0</field>
 <field rel="ManyToOneRel" to="orm.release" name="release">1</field>
 <field type="CharField" name="branch">rocko</field>
 <field type="CharField" name="dirpath">meta</field>
</object>
<object model="orm.layer_version" pk="2">
 <field rel="ManyToOneRel" to="orm.layer" name="layer">1</field>
 <field type="IntegerField" name="layer_source">0</field>
 <field rel="ManyToOneRel" to="orm.release" name="release">2</field>
 <field type="CharField" name="branch">HEAD</field>
 <field type="CharField" name="commit">HEAD</field>
 <field type="CharField" name="dirpath">meta</field>
</object>
<object model="orm.layer_version" pk="3">
 <field rel="ManyToOneRel" to="orm.layer" name="layer">1</field>
 <field type="IntegerField" name="layer_source">0</field>
 <field rel="ManyToOneRel" to="orm.release" name="release">3</field>
 <field type="CharField" name="branch">master</field>
 <field type="CharField" name="dirpath">meta</field>
</object>
```

The layer "pk" values above must be unique, and typically start at "1". The layer version "pk" values must also be unique across all layers, and typically start at "1".

### 40.4. Remote Toaster Monitoring¶

Toaster has an API that allows remote management applications to directly query the state of the Toaster server and its builds in a machine-to-machine manner. This API uses the [REST](#) interface and the transfer of JSON files. For example, you might monitor a build inside a container through well supported known HTTP ports in order to easily access a Toaster server inside the container. In this example, when you use this direct JSON API, you avoid having web page parsing against the display the user sees.

#### 40.4.1. Checking Health¶

Before you use remote Toaster monitoring, you should do a health check. To do this, ping the Toaster server using the following call to see if it is still alive:

```
http://host:port/health
```

Be sure to provide values for *host* and *port*. If the server is alive, you will get the response HTML:

```
<!DOCTYPE html>
<html lang="en">
 <head><title>Toaster Health</title></head>
 <body>0k</body>
</html>
```

#### 40.4.2. Determining Status of Builds in Progress¶

Sometimes it is useful to determine the status of a build in progress. To get the status of pending builds, use the following call:

```
http://host:port/toastergui/api/building
```

Be sure to provide values for *host* and *port*. The output is a JSON file that itemizes all builds in progress. This file includes the time in seconds since each respective build started as well as the progress of the cloning, parsing, and task execution. The following is sample output for a build in progress:

```
{ "count": 1,
 "building": [
 { "machine": "beaglebone",
 "seconds": "463.869",
 "task": "927:2384",
 "distro": "poky",
 "clone": "1:1",
 "id": 2,
 "start": "2017-09-22T09:31:44.887Z",
 "name": "20170922093200",
 "parse": "818:818",
 "project": "my_rocko",
 "target": "core-image-minimal"
 }
]
}
```

The JSON data for this query is returned in a single line. In the previous example the line has been artificially split for readability.

#### 40.4.3. Checking Status of Builds Completed¶

Once a build is completed, you get the status when you use the following call:

```
http://host:port/toastergui/api/builds
```

Be sure to provide values for *host* and *port*. The output is a JSON file that itemizes all complete builds, and includes build summary information. The following is sample output for a completed build:

```
{ "count": 1,
 "builds": [
 { "distro": "poky",
 "errors": 0,
 "machine": "beaglebone",
 "project": "my_rocko",
 "stop": "2017-09-22T09:26:36.017Z",
 "target": "quilt-native",
 "seconds": "78.193",
 "outcome": "Succeeded",
 "id": 1,
 "start": "2017-09-22T09:25:17.824Z",
 "warnings": 1,
 "name": "20170922092618"
 }
]
}
```

The JSON data for this query is returned in a single line. In the previous example the line has been artificially split for readability.

#### 40.4.4. Determining Status of a Specific Build¶

Sometimes it is useful to determine the status of a specific build. To get the status of a specific build, use the following call:

```
http://host:port/toastergui/api/build/ID
```

Be sure to provide values for *host*, *port*, and *ID*. You can find the value for *ID* from the Builds Completed query. See the "[Checking Status of Builds Completed](#)" section for more information.

The output is a JSON file that itemizes the specific build and includes build summary information. The following is sample output for a specific build:

```
{ "build":
 { "distro": "poky",
 "errors": 0,
 "machine": "beaglebone",
 "project": "my_rocko",
 "stop": "2017-09-22T09:26:36.017Z",
 "target": "quilt-native",
 "seconds": "78.193",
 "outcome": "Succeeded",
 "id": 1,
 "start": "2017-09-22T09:25:17.824Z",
 "warnings": 1,
 "name": "20170922092618",
 "cooker_log": "/opt/user/poky/build-toaster-2/tmp/log/cooker/beaglebone/build_20170922_022607.991.log"
 }
}
```

The JSON data for this query is returned in a single line. In the previous example the line has been artificially split for readability.

## 40.5. Useful Commands¶

In addition to the web user interface and the scripts that start and stop Toaster, command-line commands exist through the `manage.py` management script. You can find general documentation on `manage.py` at the [Django](#) site. However, several `manage.py` commands have been created that are specific to Toaster and are used to control configuration and back-end tasks. You can locate these commands in the [Source Directory](#) (e.g. `poky`) at `bitbake/lib/manage.py`. This section documents those commands.

### Notes

- When using `manage.py` commands given a default configuration, you must be sure that your working directory is set to the [Build Directory](#). Using `manage.py` commands from the Build Directory allows Toaster to find the `toaster.sqlite` file, which is located in the Build Directory.
- For non-default database configurations, it is possible that you can use `manage.py` commands from a directory other than the Build Directory. To do so, the `toastermain/settings.py` file must be configured to point to the correct database backend.

### 40.5.1. buildslslist¶

The `buildslslist` command lists all builds that Toaster has recorded. Access the command as follows:

```
$ bitbake/lib/toaster/manage.py buildslslist
```

The command returns a list, which includes numeric identifications, of the builds that Toaster has recorded in the current database.

You need to run the `buildslslist` command first to identify existing builds in the database before using the [builddelete](#) command. Here is an example that assumes default repository and build directory names:

```
$ cd ~/poky/build
$ python ../bitbake/lib/toaster/manage.py buildslslist
```

If your Toaster database had only one build, the above `buildslslist` command would return something like the following:

```
1: qemux86 poky core-image-minimal
```

### 40.5.2. builddelete¶

The `builddelete` command deletes data associated with a build. Access the command as follows:

```
$ bitbake/lib/toaster/manage.py builddelete build_id
```

The command deletes all the build data for the specified `build_id`. This command is useful for removing old and unused data from the database.

Prior to running the `builddelete` command, you need to get the ID associated with builds by using the [buildslslist](#) command.

### 40.5.3. perf¶

The `perf` command measures Toaster performance. Access the command as follows:

```
$ bitbake/lib/toaster/manage.py perf
```

The command is a sanity check that returns page loading times in order to identify performance problems.

### 40.5.4. checksettings¶

The `checksettings` command verifies existing Toaster settings. Access the command as follows:

```
$ bitbake/lib/toaster/manage.py checksettings
```

Toaster uses settings that are based on the database to configure the building tasks. The `checksettings` command verifies that the database settings are valid in the sense that they have the minimal information needed to start a build.

In order for the `checksettings` command to work, the database must be correctly set up and not have existing data. To be sure the database is ready, you can run the following:

```
$ bitbake/lib/toaster/manage.py syncdb
$ bitbake/lib/toaster/manage.py migrate orm
$ bitbake/lib/toaster/manage.py migrate bldcontrol
```

After running these commands, you can run the `checksettings` command.

### 40.5.5. runbuilds¶

The `runbuilds` command launches scheduled builds. Access the command as follows:

```
$ bitbake/lib/toaster/manage.py runbuilds
```

The `runbuilds` command checks if scheduled builds exist in the database and then launches them per schedule. The command returns after the builds start but before they complete. The Toaster Logging Interface records and updates the database when the builds complete.

# BitBake User Manual

## Chapter 41. Overview¶

### [41.1. Introduction](#)

### [41.2. History and Goals](#)

### [41.3. Concepts](#)

#### [41.3.1. Recipes](#)

#### [41.3.2. Configuration Files](#)

#### [41.3.3. Classes](#)

#### [41.3.4. Layers](#)

#### [41.3.5. Append Files](#)

### [41.4. Obtaining BitBake](#)

### [41.5. The BitBake Command](#)

#### [41.5.1. Usage and syntax](#)

#### [41.5.2. Examples](#)

Welcome to the BitBake User Manual. This manual provides information on the BitBake tool. The information attempts to be as independent as possible regarding systems that use BitBake, such as OpenEmbedded and the Yocto Project. In some cases, scenarios or examples within the context of a build system are used in the manual to help with understanding. For these cases, the manual clearly states the context.

## 41.1. Introduction¶

Fundamentally, BitBake is a generic task execution engine that allows shell and Python tasks to be run efficiently and in parallel while working within complex inter-task dependency constraints. One of BitBake's main users, OpenEmbedded, takes this core and builds embedded Linux software stacks using a task-oriented approach.

Conceptually, BitBake is similar to GNU Make in some regards but has significant differences:

- BitBake executes tasks according to provided metadata that builds up the tasks. Metadata is stored in recipe (`.bb`) and related recipe "append" (`.bbappend`) files, configuration (`.conf`) and underlying include (`.inc`) files, and in class (`.bbclass`) files. The metadata provides BitBake with instructions on what tasks to run and the dependencies between those tasks.
- BitBake includes a fetcher library for obtaining source code from various places such as local files, source control systems, or websites.
- The instructions for each unit to be built (e.g. a piece of software) are known as "recipe" files and contain all the information about the unit (dependencies, source file locations, checksums, description and so on).
- BitBake includes a client/server abstraction and can be used from a command line or used as a service over XML-RPC and has several different user interfaces.

## 41.2. History and Goals¶

BitBake was originally a part of the OpenEmbedded project. It was inspired by the Portage package management system used by the Gentoo Linux distribution. On December 7, 2004, OpenEmbedded project team member Chris Larson split the project into two distinct pieces:

- BitBake, a generic task executor
- OpenEmbedded, a metadata set utilized by BitBake

Today, BitBake is the primary basis of the [OpenEmbedded](#) project, which is being used to build and maintain Linux distributions such as the [Angstrom Distribution](#), and which is also being used as the build tool for Linux projects such as the [Yocto Project](#).

Prior to BitBake, no other build tool adequately met the needs of an aspiring embedded Linux distribution. All of the build systems used by traditional desktop Linux distributions lacked important functionality, and none of the ad hoc Buildroot-based systems, prevalent in the embedded space, were scalable or maintainable.

Some important original goals for BitBake were:

- Handle cross-compilation.
- Handle inter-package dependencies (build time on target architecture, build time on native architecture, and runtime).
- Support running any number of tasks within a given package, including, but not limited to, fetching upstream sources, unpacking them, patching them, configuring them, and so forth.
- Be Linux distribution agnostic for both build and target systems.

- Be architecture agnostic.
- Support multiple build and target operating systems (e.g. Cygwin, the BSDs, and so forth).
- Be self contained, rather than tightly integrated into the build machine's root filesystem.
- Handle conditional metadata on the target architecture, operating system, distribution, and machine.
- Be easy to use the tools to supply local metadata and packages against which to operate.
- Be easy to use BitBake to collaborate between multiple projects for their builds.
- Provide an inheritance mechanism to share common metadata between many packages.

Over time it became apparent that some further requirements were necessary:

- Handle variants of a base recipe (e.g. native, sdk, and multilib).
- Split metadata into layers and allow layers to enhance or override other layers.
- Allow representation of a given set of input variables to a task as a checksum. Based on that checksum, allow acceleration of builds with prebuilt components.

BitBake satisfies all the original requirements and many more with extensions being made to the basic functionality to reflect the additional requirements. Flexibility and power have always been the priorities. BitBake is highly extensible and supports embedded Python code and execution of any arbitrary tasks.

### 41.3. Concepts¶

BitBake is a program written in the Python language. At the highest level, BitBake interprets metadata, decides what tasks are required to run, and executes those tasks. Similar to GNU Make, BitBake controls how software is built. GNU Make achieves its control through "makefiles", while BitBake uses "recipes".

BitBake extends the capabilities of a simple tool like GNU Make by allowing for the definition of much more complex tasks, such as assembling entire embedded Linux distributions.

The remainder of this section introduces several concepts that should be understood in order to better leverage the power of BitBake.

#### 41.3.1. Recipes¶

BitBake Recipes, which are denoted by the file extension `.bb`, are the most basic metadata files. These recipe files provide BitBake with the following:

- Descriptive information about the package (author, homepage, license, and so on)
- The version of the recipe
- Existing dependencies (both build and runtime dependencies)
- Where the source code resides and how to fetch it
- Whether the source code requires any patches, where to find them, and how to apply them
- How to configure and compile the source code
- Where on the target machine to install the package or packages created

Within the context of BitBake, or any project utilizing BitBake as its build system, files with the `.bb` extension are referred to as recipes.

#### Note

The term "package" is also commonly used to describe recipes. However, since the same word is used to describe packaged output from a project, it is best to maintain a single descriptive term - "recipes". Put another way, a single "recipe" file is quite capable of generating a number of related but separately installable "packages". In fact, that ability is fairly common.

#### 41.3.2. Configuration Files¶

Configuration files, which are denoted by the `.conf` extension, define various configuration variables that govern the project's build process. These files fall into several areas that define machine configuration options, distribution configuration options, compiler tuning options, general common configuration options, and user configuration options. The main configuration file is the sample `bitbake.conf` file, which is located within the BitBake source tree `conf` directory.

#### 41.3.3. Classes¶

Class files, which are denoted by the `.bbclass` extension, contain information that is useful to share between metadata files. The BitBake source tree currently comes with one class metadata file called `base.bbclass`. You can find this file in the `classes` directory. The `base.bbclass` class file is special since it is always included automatically for all recipes and classes. This class contains definitions for standard basic tasks such as fetching, unpacking, configuring (empty by default), compiling (runs any Makefile present), installing (empty by default) and packaging (empty by default). These tasks are often overridden or extended by other classes added during the project development process.

#### 41.3.4. Layers¶



Layers allow you to isolate different types of customizations from each other. While you might find it tempting to keep everything in one layer when working on a single project, the more modular you organize your metadata, the easier it is to cope with future changes.

To illustrate how you can use layers to keep things modular, consider customizations you might make to support a specific target machine. These types of customizations typically reside in a special layer, rather than a general layer, called a Board Support Package (BSP) Layer. Furthermore, the machine customizations should be isolated from recipes and metadata that support a new GUI environment, for example. This situation gives you a couple of layers: one for the machine configurations and one for the GUI environment. It is important to understand, however, that the BSP layer can still make machine-specific additions to recipes within the GUI environment layer without polluting the GUI layer itself with those machine-specific changes. You can accomplish this through a recipe that is a BitBake append (`.bbappend`) file.

#### 41.3.5. Append Files¶

Append files, which are files that have the `.bbappend` file extension, extend or override information in an existing recipe file.

BitBake expects every append file to have a corresponding recipe file. Furthermore, the append file and corresponding recipe file must use the same root filename. The filenames can differ only in the file type suffix used (e.g. `formfactor_0.0.bb` and `formfactor_0.0.bbappend`).

Information in append files extends or overrides the information in the underlying, similarly-named recipe files.

When you name an append file, you can use the `"%"` wildcard character to allow for matching recipe names. For example, suppose you have an append file named as follows:

```
busybox_1.21.%.bbappend
```

That append file would match any `busybox_1.21.x.bb` version of the recipe. So, the append file would match the following recipe names:

```
busybox_1.21.1.bb
busybox_1.21.2.bb
busybox_1.21.3.bb
```

### Important

The use of the `"%"` character is limited in that it only works directly in front of the `.bbappend` portion of the append file's name. You cannot use the wildcard character in any other location of the name.

If the `busybox` recipe was updated to `busybox_1.3.0.bb`, the append name would not match. However, if you named the append file `busybox_1.%.bbappend`, then you would have a match.

In the most general case, you could name the append file something as simple as `busybox_%.bbappend` to be entirely version independent.

#### 41.4. Obtaining BitBake¶

You can obtain BitBake several different ways:

- **Cloning BitBake:** Using Git to clone the BitBake source code repository is the recommended method for obtaining BitBake. Cloning the repository makes it easy to get bug fixes and have access to stable branches and the master branch. Once you have cloned BitBake, you should use the latest stable branch for development since the master branch is for BitBake development and might contain less stable changes.

You usually need a version of BitBake that matches the metadata you are using. The metadata is generally backwards compatible but not forward compatible.

Here is an example that clones the BitBake repository:

```
$ git clone git://git.openembedded.org/bitbake
```

This command clones the BitBake Git repository into a directory called `bitbake`. Alternatively, you can designate a directory after the `git clone` command if you want to call the new directory something other than `bitbake`. Here is an example that names the directory `bbdev`:

```
$ git clone git://git.openembedded.org/bitbake bbdev
```

- **Installation using your Distribution Package Management System:** This method is not recommended because the BitBake version that is provided by your distribution, in most cases, is several releases behind a snapshot of the BitBake repository.
- **Taking a snapshot of BitBake:** Downloading a snapshot of BitBake from the source code repository gives you access to a known branch or release of BitBake.

### Note

Cloning the Git repository, as described earlier, is the preferred method for getting BitBake. Cloning the repository makes it easier to update as patches are added to the stable branches.

The following example downloads a snapshot of BitBake version 1.17.0:

```
$ wget http://git.openembedded.org/bitbake/snapshot/bitbake-1.17.0.tar.gz
$ tar xpvf bitbake-1.17.0.tar.gz
```

After extraction of the tarball using the tar utility, you have a directory entitled `bitbake-1.17.0`.

- **Using the BitBake that Comes With Your Build Checkout:** A final possibility for getting a copy of BitBake is that it already comes with your checkout of a larger BitBake-based build system, such as Poky. Rather than manually checking out individual layers and gluing them together yourself, you can check out an entire build system. The checkout will already include a version of BitBake that has been thoroughly tested for compatibility with the other components. For information on how to check out a particular BitBake-based build system, consult that build system's supporting documentation.

## 41.5. The BitBake Command¶

The `bitbake` command is the primary interface to the BitBake tool. This section presents the BitBake command syntax and provides several execution examples.

### 41.5.1. Usage and syntax¶

Following is the usage and syntax for BitBake:

```
$ bitbake -h
Usage: bitbake [options] [recipeName/target recipe:do_task ...]

 Executes the specified task (default is 'build') for a given set of target recipes (.bb files).
 It is assumed there is a conf/bblayers.conf available in cwd or in BBPATH which
 will provide the layer, BBFILES and other configuration information.

Options:
--version show program's version number and exit
-h, --help show this help message and exit
-b BUILDFILE, --buildfile=BUILDFILE
 Execute tasks from a specific .bb recipe directly.
 WARNING: Does not handle any dependencies from other
 recipes.
-k, --continue Continue as much as possible after an error. While the
 target that failed and anything depending on it cannot
 be built, as much as possible will be built before
 stopping.
-f, --force Force the specified targets/task to run (invalidating
 any existing stamp file).
-c CMD, --cmd=CMD Specify the task to execute. The exact options
 available depend on the metadata. Some examples might
 be 'compile' or 'populate_sysroot' or 'listtasks' may
 give a list of the tasks available.
-C INVALIDATE_STAMP, --clear-stamp=INVALIDATE_STAMP
 Invalidate the stamp for the specified task such as
 'compile' and then run the default task for the
 specified target(s).
-r PREFILE, --read=PREFILE
 Read the specified file before bitbake.conf.
-R POSTFILE, --postread=POSTFILE
 Read the specified file after bitbake.conf.
-v, --verbose Enable tracing of shell tasks (with 'set -x'). Also
 print bb.note(...) messages to stdout (in addition to
 writing them to ${T}/log.do_<task>).
-D, --debug Increase the debug level. You can specify this more
 than once. -D sets the debug level to 1, where only
 bb.debug(1, ...) messages are printed to stdout; -DD
 sets the debug level to 2, where both bb.debug(1, ...)
 and bb.debug(2, ...) messages are printed; etc.
 Without -D, no debug messages are printed. Note that
 -D only affects output to stdout. All debug messages
 are written to ${T}/log.do_taskname, regardless of the
 debug level.
-q, --quiet Output less log message data to the terminal. You can
 specify this more than once.
-n, --dry-run Don't execute, just go through the motions.
-S SIGNATURE_HANDLER, --dump-signatures=SIGNATURE_HANDLER
 Dump out the signature construction information, with
 no task execution. The SIGNATURE_HANDLER parameter is
 passed to the handler. Two common values are none and
 printdiff but the handler may define more/less. none
 means only dump the signature, printdiff means compare
 the dumped signature with the cached one.
-p, --parse-only Quit after parsing the BB recipes.
-s, --show-versions Show current and preferred versions of all recipes.
-e, --environment Show the global or per-recipe environment complete
 with information about where variables were
 set/changed.
-g, --graphviz Save dependency tree information for the specified
 targets in the dot syntax.
-I EXTRA_ASSUME_PROVIDED, --ignore-deps=EXTRA_ASSUME_PROVIDED
 Assume these dependencies don't exist and are already
 provided (equivalent to ASSUME_PROVIDED). Useful to
 make dependency graphs more appealing
-l DEBUG_DOMAINS, --log-domains=DEBUG_DOMAINS
 Show debug logging for the specified logging domains
-P, --profile Profile the command and save reports.
-u UI, --ui=UI The user interface to use (knotty, ncurses or taskexp
 - default knotty).
--token=XMLRPC_TOKEN Specify the connection token to be used when
 connecting to a remote server.
--revisions-changed Set the exit code depending on whether upstream
 floating revisions have changed or not.
--server-only Run bitbake without a UI, only starting a server
 (cooker) process.
-B BIND, --bind=BIND The name/address for the bitbake xmlrpc server to bind
 to.
-T SERVER_TIMEOUT, --idle-timeout=SERVER_TIMEOUT
 Set timeout to unload bitbake server due to
 inactivity, set to -1 means no unload, default:
```

|                                                          |                                                                                                                             |
|----------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
|                                                          | Environment variable <code>BB_SERVER_TIMEOUT</code> .                                                                       |
| <code>--no-setscene</code>                               | Do not run any setscene tasks. <code>sstate</code> will be ignored and everything needed, built.                            |
| <code>--setscene-only</code>                             | Only run setscene tasks, don't run any real tasks.                                                                          |
| <code>--remote-server=REMOTE_SERVER</code>               | Connect to the specified server.                                                                                            |
| <code>-m, --kill-server</code>                           | Terminate any running bitbake server.                                                                                       |
| <code>--observe-only</code>                              | Connect to a server as an observing-only client.                                                                            |
| <code>--status-only</code>                               | Check the status of the remote bitbake server.                                                                              |
| <code>-w WRITEEVENTLOG, --write-log=WRITEEVENTLOG</code> | Writes the event log of the build to a bitbake event json file. Use '' (empty string) to assign the name automatically.     |
| <code>--runall=RUNALL</code>                             | Run the specified task for any recipe in the taskgraph of the specified target (even if it wouldn't otherwise have run).    |
| <code>--runonly=RUNONLY</code>                           | Run only the specified task within the taskgraph of the specified targets (and any task dependencies those tasks may have). |

## 41.5.2. Examples¶

This section presents some examples showing how to use BitBake.

### 41.5.2.1. Executing a Task Against a Single Recipe¶

Executing tasks for a single recipe file is relatively simple. You specify the file in question, and BitBake parses it and executes the specified task. If you do not specify a task, BitBake executes the default task, which is "build". BitBake obeys inter-task dependencies when doing so.

The following command runs the build task, which is the default task, on the `foo_1.0.bb` recipe file:

```
$ bitbake -b foo_1.0.bb
```

The following command runs the clean task on the `foo.bb` recipe file:

```
$ bitbake -b foo.bb -c clean
```

### Note

The "-b" option explicitly does not handle recipe dependencies. Other than for debugging purposes, it is instead recommended that you use the syntax presented in the next section.

### 41.5.2.2. Executing Tasks Against a Set of Recipe Files¶

There are a number of additional complexities introduced when one wants to manage multiple `.bb` files. Clearly there needs to be a way to tell BitBake what files are available and, of those, which you want to execute. There also needs to be a way for each recipe to express its dependencies, both for build-time and runtime. There must be a way for you to express recipe preferences when multiple recipes provide the same functionality, or when there are multiple versions of a recipe.

The `bitbake` command, when not using "--buildfile" or "-b" only accepts a "PROVIDES". You cannot provide anything else. By default, a recipe file generally "PROVIDES" its "packagename" as shown in the following example:

```
$ bitbake foo
```

This next example "PROVIDES" the package name and also uses the "-c" option to tell BitBake to just execute the `do_clean` task:

```
$ bitbake -c clean foo
```

### 41.5.2.3. Executing a List of Task and Recipe Combinations¶

The BitBake command line supports specifying different tasks for individual targets when you specify multiple targets. For example, suppose you had two targets (or recipes) `myfirstrecipe` and `mysecondrecipe` and you needed BitBake to run `taskA` for the first recipe and `taskB` for the second recipe:

```
$ bitbake myfirstrecipe:do_taskA mysecondrecipe:do_taskB
```

### 41.5.2.4. Generating Dependency Graphs¶

BitBake is able to generate dependency graphs using the `dot` syntax. You can convert these graphs into images using the `dot` tool from [Graphviz](#).

When you generate a dependency graph, BitBake writes three files to the current working directory:

- **`recipe-depends.dot`**: Shows dependencies between recipes (i.e. a collapsed version of `task-depends.dot`).
- **`task-depends.dot`**: Shows dependencies between tasks. These dependencies match BitBake's internal task execution list.
- **`pn-buildlist`**: Shows a simple list of targets that are to be built.

To stop depending on common depends, use the "-I" depend option and BitBake omits them from the graph. Leaving this information out can produce more readable graphs. This way, you can remove from the graph `DEPENDS` from inherited classes such as `base.bbclass`.

Here are two examples that create dependency graphs. The second example omits depends common in OpenEmbedded from the graph:

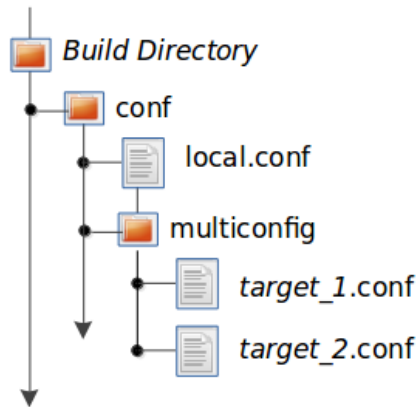
```
$ bitbake -g foo

$ bitbake -g -I virtual/kernel -I eglibc foo
```

#### 41.5.2.5. Executing a Multiple Configuration Build¶

BitBake is able to build multiple images or packages using a single command where the different targets require different configurations (multiple configuration builds). Each target, in this scenario, is referred to as a "multiconfig".

To accomplish a multiple configuration build, you must define each target's configuration separately using a parallel configuration file in the build directory. The location for these multiconfig configuration files is specific. They must reside in the current build directory in a sub-directory of `conf` named `multiconfig`. Following is an example for two separate targets:



The reason for this required file hierarchy is because the `BBPATH` variable is not constructed until the layers are parsed. Consequently, using the configuration file as a pre-configuration file is not possible unless it is located in the current working directory.

Minimally, each configuration file must define the machine and the temporary directory BitBake uses for the build. Suggested practice dictates that you do not overlap the temporary directories used during the builds.

Aside from separate configuration files for each target, you must also enable BitBake to perform multiple configuration builds. Enabling is accomplished by setting the `BBMULTICONFIG` variable in the `local.conf` configuration file. As an example, suppose you had configuration files for `target1` and `target2` defined in the build directory. The following statement in the `local.conf` file both enables BitBake to perform multiple configuration builds and specifies the two extra multiconfigs:

```
BBMULTICONFIG = "target1 target2"
```

Once the target configuration files are in place and BitBake has been enabled to perform multiple configuration builds, use the following command form to start the builds:

```
$ bitbake [mc:multiconfigname:]target [[mc:multiconfigname:]target] ...]
```

Here is an example for two extra multiconfigs: `target1` and `target2`:

```
$ bitbake mc::target mc:target1:target mc:target2:target
```

#### 41.5.2.6. Enabling Multiple Configuration Build Dependencies¶

Sometimes dependencies can exist between targets (multiconfigs) in a multiple configuration build. For example, suppose that in order to build an image for a particular architecture, the root filesystem of another build for a different architecture needs to exist. In other words, the image for the first multiconfig depends on the root filesystem of the second multiconfig. This dependency is essentially that the task in the recipe that builds one multiconfig is dependent on the completion of the task in the recipe that builds another multiconfig.

To enable dependencies in a multiple configuration build, you must declare the dependencies in the recipe using the following statement form:

```
task_or_package[mcdepends] = "mc:from_multiconfig:to_multiconfig:recipe_name:task_on_which_to_depend"
```

To better show how to use this statement, consider an example with two multiconfigs: `target1` and `target2`:

```
image_task[mcdepends] = "mc:target1:target2:image2:rootfs_task"
```

In this example, the `from_multiconfig` is "target1" and the `to_multiconfig` is "target2". The task on which the image whose recipe contains `image_task` depends on the completion of the `rootfs_task` used to build out `image2`, which is associated with the "target2" multiconfig.

Once you set up this dependency, you can build the "target1" multiconfig using a BitBake command as follows:

```
$ bitbake mc:target1:image1
```

This command executes all the tasks needed to create `image1` for the "target1" multiconfig. Because of the dependency, BitBake also executes through the `rootfs_task` for the "target2" multiconfig build.

Having a recipe depend on the root filesystem of another build might not seem that useful. Consider this change to the statement in the `image1` recipe:

```
image_task[mcdepends] = "mc:target1:target2:image2:image_task"
```

In this case, BitBake must create `image2` for the "target2" build since the "target1" build depends on it.

Because "target1" and "target2" are enabled for multiple configuration builds and have separate configuration files, BitBake places the artifacts for each build in the respective temporary build directories.

## Chapter 42. Execution¶

### [42.1. Parsing the Base Configuration Metadata](#)

### [42.2. Locating and Parsing Recipes](#)

### [42.3. Providers](#)

### [42.4. Preferences](#)

### [42.5. Dependencies](#)

### [42.6. The Task List](#)

### [42.7. Executing Tasks](#)

### [42.8. Checksums \(Signatures\)](#)

### [42.9. Setscene](#)

The primary purpose for running BitBake is to produce some kind of output such as a single installable package, a kernel, a software development kit, or even a full, board-specific bootable Linux image, complete with bootloader, kernel, and root filesystem. Of course, you can execute the `bitbake` command with options that cause it to execute single tasks, compile single recipe files, capture or clear data, or simply return information about the execution environment.

This chapter describes BitBake's execution process from start to finish when you use it to create an image. The execution process is launched using the following command form:

```
$ bitbake target
```

For information on the BitBake command and its options, see "[The BitBake Command](#)" section.

## Note

Prior to executing BitBake, you should take advantage of available parallel thread execution on your build host by setting the `BB_NUMBER_THREADS` variable in your project's `local.conf` configuration file.

A common method to determine this value for your build host is to run the following:

```
$ grep processor /proc/cpuinfo
```

This command returns the number of processors, which takes into account hyper-threading. Thus, a quad-core build host with hyper-threading most likely shows eight processors, which is the value you would then assign to `BB_NUMBER_THREADS`.

A possibly simpler solution is that some Linux distributions (e.g. Debian and Ubuntu) provide the `nproc` command.

## 42.1. Parsing the Base Configuration Metadata¶

The first thing BitBake does is parse base configuration metadata. Base configuration metadata consists of your project's `bblayers.conf` file to determine what layers BitBake needs to recognize, all necessary `layer.conf` files (one from each layer), and `bitbake.conf`. The data itself is of various types:

- **Recipes:** Details about particular pieces of software.
- **Class Data:** An abstraction of common build information (e.g. how to build a Linux kernel).
- **Configuration Data:** Machine-specific settings, policy decisions, and so forth. Configuration data acts as the glue to bind everything together.

The `layer.conf` files are used to construct key variables such as `BBPATH` and `BBFILES`. `BBPATH` is used to search for configuration and class files under the `conf` and `classes` directories, respectively. `BBFILES` is used to locate both recipe and recipe append files (`.bb` and `.bbappend`). If there is no `bblayers.conf` file, it is assumed the user has set the `BBPATH` and `BBFILES` directly in the environment.

Next, the `bitbake.conf` file is located using the `BBPATH` variable that was just constructed. The `bitbake.conf` file may also include other configuration files using the `include` or `require` directives.

Prior to parsing configuration files, Bitbake looks at certain variables, including:

- `BB_ENV_WHITELIST`
- `BB_ENV_EXTRAWHITE`
- `BB_PRESERVE_ENV`
- `BB_ORIGENV`
- `BITBAKE_UI`

The first four variables in this list relate to how BitBake treats shell environment variables during task execution. By default, BitBake cleans the environment variables and provides tight control over the shell execution environment. However, through the use of these first four variables, you can apply your control regarding the environment variables allowed to be used by BitBake in the shell during

execution of tasks. See the "[Passing Information Into the Build Task Environment](#)" section and the information about these variables in the variable glossary for more information on how they work and on how to use them.

The base configuration metadata is global and therefore affects all recipes and tasks that are executed.

BitBake first searches the current working directory for an optional `conf/bblayers.conf` configuration file. This file is expected to contain a `BBLAYERS` variable that is a space-delimited list of 'layer' directories. Recall that if BitBake cannot find a `bblayers.conf` file, then it is assumed the user has set the `BBPATH` and `BBFILES` variables directly in the environment.

For each directory (layer) in this list, a `conf/layer.conf` file is located and parsed with the `LAYERDIR` variable being set to the directory where the layer was found. The idea is these files automatically set up `BBPATH` and other variables correctly for a given build directory.

BitBake then expects to find the `conf/bitbake.conf` file somewhere in the user-specified `BBPATH`. That configuration file generally has include directives to pull in any other metadata such as files specific to the architecture, the machine, the local environment, and so forth.

Only variable definitions and include directives are allowed in BitBake `.conf` files. Some variables directly influence BitBake's behavior. These variables might have been set from the environment depending on the environment variables previously mentioned or set in the configuration files. The "[Variables Glossary](#)" chapter presents a full list of variables.

After parsing configuration files, BitBake uses its rudimentary inheritance mechanism, which is through class files, to inherit some standard classes. BitBake parses a class when the `inherit` directive responsible for getting that class is encountered.

The `base.bbclass` file is always included. Other classes that are specified in the configuration using the `INHERIT` variable are also included. BitBake searches for class files in a `classes` subdirectory under the paths in `BBPATH` in the same way as configuration files.

A good way to get an idea of the configuration files and the class files used in your execution environment is to run the following BitBake command:

```
$ bitbake -e > mybb.log
```

Examining the top of the `mybb.log` shows you the many configuration files and class files used in your execution environment.

## Note

You need to be aware of how BitBake parses curly braces. If a recipe uses a closing curly brace within the function and the character has no leading spaces, BitBake produces a parsing error. If you use a pair of curly braces in a shell function, the closing curly brace must not be located at the start of the line without leading spaces.

Here is an example that causes BitBake to produce a parsing error:

```
fakeroot create_shar() {
 cat << "EOF" > ${SDK_DEPLOY}/${TOOLCHAIN_OUTPUTNAME}.sh
 usage()
 {
 echo "test"
 ##### The following "}" at the start of the line causes a parsing error #####
 }
 EOF
}
```

Writing the recipe this way avoids the error:

```
fakeroot create_shar() {
 cat << "EOF" > ${SDK_DEPLOY}/${TOOLCHAIN_OUTPUTNAME}.sh
 usage()
 {
 echo "test"
 #####The following "}" with a leading space at the start of the line avoids the error #####
 }
 EOF
}
```

## 42.2. Locating and Parsing Recipes¶

During the configuration phase, BitBake will have set `BBFILES`. BitBake now uses it to construct a list of recipes to parse, along with any append files (`.bbappend`) to apply. `BBFILES` is a space-separated list of available files and supports wildcards. An example would be:

```
BBFILES = "/path/to/bbfiles/*.bb /path/to/appends/*.bbappend"
```

BitBake parses each recipe and append file located with `BBFILES` and stores the values of various variables into the datastore.

## Note

Append files are applied in the order they are encountered in `BBFILES`.

For each file, a fresh copy of the base configuration is made, then the recipe is parsed line by line. Any `inherit` statements cause BitBake to find and then parse class files (`.bbclass`) using `BBPATH` as the search path. Finally, BitBake parses in order any append files found in `BBFILES`.

One common convention is to use the recipe filename to define pieces of metadata. For example, in `bitbake.conf` the recipe name and version are used to set the variables `PN` and `PV`:

```
PN = "${@bb.parse.BBHandler.vars_from_file(d.getVar('FILE', False),d)[0] or 'defaultpkgname'}"
PV = "${@bb.parse.BBHandler.vars_from_file(d.getVar('FILE', False),d)[1] or '1.0'}"
```

In this example, a recipe called "something\_1.2.3.bb" would set `PN` to "something" and `PV` to "1.2.3".

By the time parsing is complete for a recipe, BitBake has a list of tasks that the recipe defines and a set of data consisting of keys and values as well as dependency information about the tasks.

BitBake does not need all of this information. It only needs a small subset of the information to make decisions about the recipe. Consequently, BitBake caches the values in which it is interested and does not store the rest of the information. Experience has shown it is faster to re-parse the metadata than to try and write it out to the disk and then reload it.

Where possible, subsequent BitBake commands reuse this cache of recipe information. The validity of this cache is determined by first computing a checksum of the base configuration data (see `BB_HASHCONFIG_WHITELIST`) and then checking if the checksum matches. If that checksum matches what is in the cache and the recipe and class files have not changed, BitBake is able to use the cache. BitBake then reloads the cached information about the recipe instead of reparsing it from scratch.

Recipe file collections exist to allow the user to have multiple repositories of `.bb` files that contain the same exact package. For example, one could easily use them to make one's own local copy of an upstream repository, but with custom modifications that one does not want upstream. Here is an example:

```
BBFILES = "/stuff/openembedded/*/*.bb /stuff/openembedded.modified/*/*.bb"
BBFILE_COLLECTIONS = "upstream local"
BBFILE_PATTERN_upstream = "^/stuff/openembedded/"
BBFILE_PATTERN_local = "^/stuff/openembedded.modified/"
BBFILE_PRIORITY_upstream = "5"
BBFILE_PRIORITY_local = "10"
```

## Note

The layers mechanism is now the preferred method of collecting code. While the collections code remains, its main use is to set layer priorities and to deal with overlap (conflicts) between layers.

## 42.3. Providers

Assuming BitBake has been instructed to execute a target and that all the recipe files have been parsed, BitBake starts to figure out how to build the target. BitBake looks through the `PROVIDES` list for each of the recipes. A `PROVIDES` list is the list of names by which the recipe can be known. Each recipe's `PROVIDES` list is created implicitly through the recipe's `PN` variable and explicitly through the recipe's `PROVIDES` variable, which is optional.

When a recipe uses `PROVIDES`, that recipe's functionality can be found under an alternative name or names other than the implicit `PN` name. As an example, suppose a recipe named `keyboard_1.0.bb` contained the following:

```
PROVIDES += "fullkeyboard"
```

The `PROVIDES` list for this recipe becomes "keyboard", which is implicit, and "fullkeyboard", which is explicit. Consequently, the functionality found in `keyboard_1.0.bb` can be found under two different names.

## 42.4. Preferences

The `PROVIDES` list is only part of the solution for figuring out a target's recipes. Because targets might have multiple providers, BitBake needs to prioritize providers by determining provider preferences.

A common example in which a target has multiple providers is "virtual/kernel", which is on the `PROVIDES` list for each kernel recipe. Each machine often selects the best kernel provider by using a line similar to the following in the machine configuration file:

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

The default `PREFERRED_PROVIDER` is the provider with the same name as the target. BitBake iterates through each target it needs to build and resolves them and their dependencies using this process.

Understanding how providers are chosen is made complicated by the fact that multiple versions might exist for a given provider. BitBake defaults to the highest version of a provider. Version comparisons are made using the same method as Debian. You can use the `PREFERRED_VERSION` variable to specify a particular version. You can influence the order by using the `DEFAULT_PREFERENCE` variable.

By default, files have a preference of "0". Setting `DEFAULT_PREFERENCE` to "-1" makes the recipe unlikely to be used unless it is explicitly referenced. Setting `DEFAULT_PREFERENCE` to "1" makes it likely the recipe is used. `PREFERRED_VERSION` overrides any `DEFAULT_PREFERENCE` setting. `DEFAULT_PREFERENCE` is often used to mark newer and more experimental recipe versions until they have undergone sufficient testing to be considered stable.

When there are multiple "versions" of a given recipe, BitBake defaults to selecting the most recent version, unless otherwise specified. If the recipe in question has a `DEFAULT_PREFERENCE` set lower than the other recipes (default is 0), then it will not be selected. This allows the person or persons maintaining the repository of recipe files to specify their preference for the default selected version. Additionally, the user can specify their preferred version.

If the first recipe is named `a_1.1.bb`, then the `PN` variable will be set to "a", and the `PV` variable will be set to 1.1.

Thus, if a recipe named `a_1.2.bb` exists, BitBake will choose 1.2 by default. However, if you define the following variable in a `.conf` file that BitBake parses, you can change that preference:



```
PREFERRED_VERSION_a = "1.1"
```

## Note

It is common for a recipe to provide two versions -- a stable, numbered (and preferred) version, and a version that is automatically checked out from a source code repository that is considered more "bleeding edge" but can be selected only explicitly.

For example, in the OpenEmbedded codebase, there is a standard, versioned recipe file for BusyBox, `busybox_1.22.1.bb`, but there is also a Git-based version, `busybox_git.bb`, which explicitly contains the line

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

to ensure that the numbered, stable version is always preferred unless the developer selects otherwise.

## 42.5. Dependencies¶

Each target BitBake builds consists of multiple tasks such as `fetch`, `unpack`, `patch`, `configure`, and `compile`. For best performance on multi-core systems, BitBake considers each task as an independent entity with its own set of dependencies.

Dependencies are defined through several variables. You can find information about variables BitBake uses in the [Variables Glossary](#) near the end of this manual. At a basic level, it is sufficient to know that BitBake uses the `DEPENDS` and `RDEPENDS` variables when calculating dependencies.

For more information on how BitBake handles dependencies, see the "[Dependencies](#)" section.

## 42.6. The Task List¶

Based on the generated list of providers and the dependency information, BitBake can now calculate exactly what tasks it needs to run and in what order it needs to run them. The "[Executing Tasks](#)" section has more information on how BitBake chooses which task to execute next.

The build now starts with BitBake forking off threads up to the limit set in the `BB_NUMBER_THREADS` variable. BitBake continues to fork threads as long as there are tasks ready to run, those tasks have all their dependencies met, and the thread threshold has not been exceeded.

It is worth noting that you can greatly speed up the build time by properly setting the `BB_NUMBER_THREADS` variable.

As each task completes, a timestamp is written to the directory specified by the `STAMP` variable. On subsequent runs, BitBake looks in the build directory within `tmp/stamps` and does not rerun tasks that are already completed unless a timestamp is found to be invalid. Currently, invalid timestamps are only considered on a per recipe file basis. So, for example, if the configure stamp has a timestamp greater than the compile timestamp for a given target, then the compile task would rerun. Running the compile task again, however, has no effect on other providers that depend on that target.

The exact format of the stamps is partly configurable. In modern versions of BitBake, a hash is appended to the stamp so that if the configuration changes, the stamp becomes invalid and the task is automatically rerun. This hash, or signature used, is governed by the signature policy that is configured (see the "[Checksums \(Signatures\)](#)" section for information). It is also possible to append extra metadata to the stamp using the `[stamp-extra-info]` task flag. For example, OpenEmbedded uses this flag to make some tasks machine-specific.

## Note

Some tasks are marked as "nostamp" tasks. No timestamp file is created when these tasks are run. Consequently, "nostamp" tasks are always rerun.

For more information on tasks, see the "[Tasks](#)" section.

## 42.7. Executing Tasks¶

Tasks can be either a shell task or a Python task. For shell tasks, BitBake writes a shell script to `${T}/run.do_taskname.pid` and then executes the script. The generated shell script contains all the exported variables, and the shell functions with all variables expanded. Output from the shell script goes to the file `${T}/log.do_taskname.pid`. Looking at the expanded shell functions in the run file and the output in the log files is a useful debugging technique.

For Python tasks, BitBake executes the task internally and logs information to the controlling terminal. Future versions of BitBake will write the functions to files similar to the way shell tasks are handled. Logging will be handled in a way similar to shell tasks as well.

The order in which BitBake runs the tasks is controlled by its task scheduler. It is possible to configure the scheduler and define custom implementations for specific use cases. For more information, see these variables that control the behavior:

- `BB_SCHEDULER`
- `BB_SCHEDULERS`

It is possible to have functions run before and after a task's main function. This is done using the `[prefuncs]` and `[postfuncs]` flags of the task that lists the functions to run.



## 42.8. Checksums (Signatures)

A checksum is a unique signature of a task's inputs. The signature of a task can be used to determine if a task needs to be run. Because it is a change in a task's inputs that triggers running the task, BitBake needs to detect all the inputs to a given task. For shell tasks, this turns out to be fairly easy because BitBake generates a "run" shell script for each task and it is possible to create a checksum that gives you a good idea of when the task's data changes.

To complicate the problem, some things should not be included in the checksum. First, there is the actual specific build path of a given task - the working directory. It does not matter if the working directory changes because it should not affect the output for target packages. The simplistic approach for excluding the working directory is to set it to some fixed value and create the checksum for the "run" script. BitBake goes one step better and uses the `BB_HASHBASE_WHITELIST` variable to define a list of variables that should never be included when generating the signatures.

Another problem results from the "run" scripts containing functions that might or might not get called. The incremental build solution contains code that figures out dependencies between shell functions. This code is used to prune the "run" scripts down to the minimum set, thereby alleviating this problem and making the "run" scripts much more readable as a bonus.

So far we have solutions for shell scripts. What about Python tasks? The same approach applies even though these tasks are more difficult. The process needs to figure out what variables a Python function accesses and what functions it calls. Again, the incremental build solution contains code that first figures out the variable and function dependencies, and then creates a checksum for the data used as the input to the task.

Like the working directory case, situations exist where dependencies should be ignored. For these cases, you can instruct the build process to ignore a dependency by using a line like the following:

```
PACKAGE_ARCHS[vardepsexclude] = "MACHINE"
```

This example ensures that the `PACKAGE_ARCHS` variable does not depend on the value of `MACHINE`, even if it does reference it.

Equally, there are cases where we need to add dependencies BitBake is not able to find. You can accomplish this by using a line like the following:

```
PACKAGE_ARCHS[vardeps] = "MACHINE"
```

This example explicitly adds the `MACHINE` variable as a dependency for `PACKAGE_ARCHS`.

Consider a case with in-line Python, for example, where BitBake is not able to figure out dependencies. When running in debug mode (i.e. using `-DDD`), BitBake produces output when it discovers something for which it cannot figure out dependencies.

Thus far, this section has limited discussion to the direct inputs into a task. Information based on direct inputs is referred to as the "basehash" in the code. However, there is still the question of a task's indirect inputs - the things that were already built and present in the build directory. The checksum (or signature) for a particular task needs to add the hashes of all the tasks on which the particular task depends. Choosing which dependencies to add is a policy decision. However, the effect is to generate a master checksum that combines the basehash and the hashes of the task's dependencies.

At the code level, there are a variety of ways both the basehash and the dependent task hashes can be influenced. Within the BitBake configuration file, we can give BitBake some extra information to help it construct the basehash. The following statement effectively results in a list of global variable dependency excludes - variables never included in any checksum. This example uses variables from OpenEmbedded to help illustrate the concept:

```
BB_HASHBASE_WHITELIST ?= "TMPDIR FILE_PATH PWD BB_TASKHASH BBPATH DL_DIR \
 SSTATE_DIR THISDIR FILESEXTRAPATHS FILE_DIRNAME HOME LOGNAME SHELL TERM \
 USER FILESPATH STAGING_DIR_HOST STAGING_DIR_TARGET COREBASE PRSERV_HOST \
 PRSERV_DUMPDIR PRSERV_DUMPPFILE PRSERV_LOCKDOWN PARALLEL_MAKE \
 CCACHE_DIR EXTERNAL_TOOLCHAIN CCACHE CCACHE_DISABLE LICENSE_PATH SDKPKGSUFFIX"
```

The previous example excludes the work directory, which is part of `TMPDIR`.

The rules for deciding which hashes of dependent tasks to include through dependency chains are more complex and are generally accomplished with a Python function. The code in `meta/lib/oe/ssstatesig.py` shows two examples of this and also illustrates how you can insert your own policy into the system if so desired. This file defines the two basic signature generators OpenEmbedded-Core uses: "OEBasic" and "OEBasicHash". By default, there is a dummy "noop" signature handler enabled in BitBake. This means that behavior is unchanged from previous versions. OE-Core uses the "OEBasicHash" signature handler by default through this setting in the `bitbake.conf` file:

```
BB_SIGNATURE_HANDLER ?= "OEBasicHash"
```

The "OEBasicHash" `BB_SIGNATURE_HANDLER` is the same as the "OEBasic" version but adds the task hash to the stamp files. This results in any metadata change that changes the task hash, automatically causing the task to be run again. This removes the need to bump `PR` values, and changes to metadata automatically ripple across the build.

It is also worth noting that the end result of these signature generators is to make some dependency and hash information available to the build. This information includes:

- `BB_BASEHASH_task-taskname`: The base hashes for each task in the recipe.
- `BB_BASEHASH_filename:taskname`: The base hashes for each dependent task.
- `BBHASHDEPS_filename:taskname`: The task dependencies for each task.
- `BB_TASKHASH`: The hash of the currently running task.

It is worth noting that BitBake's "-S" option lets you debug BitBake's processing of signatures. The options passed to -S allow different debugging modes to be used, either using BitBake's own debug functions or possibly those defined in the metadata/signature handler itself. The simplest parameter to pass is "none", which causes a set of signature information to be written out into `STAMPS_DIR` corresponding to the targets specified. The other currently available parameter is "printdiff", which causes BitBake to try to establish the closest signature match it can (e.g. in the `sstate` cache) and then run `bitbake-diffsigs` over the matches to determine the stamps and delta where these two stamp trees diverge.

**Note**

It is likely that future versions of BitBake will provide other signature handlers triggered through additional "-S" parameters.

You can find more information on checksum metadata in the "[Task Checksums and Setscene](#)" section.

**42.9. Setscene¶**

The setscene process enables BitBake to handle "pre-built" artifacts. The ability to handle and reuse these artifacts allows BitBake the luxury of not having to build something from scratch every time. Instead, BitBake can use, when possible, existing build artifacts.

BitBake needs to have reliable data indicating whether or not an artifact is compatible. Signatures, described in the previous section, provide an ideal way of representing whether an artifact is compatible. If a signature is the same, an object can be reused.

If an object can be reused, the problem then becomes how to replace a given task or set of tasks with the pre-built artifact. BitBake solves the problem with the "setscene" process.

When BitBake is asked to build a given target, before building anything, it first asks whether cached information is available for any of the targets it's building, or any of the intermediate targets. If cached information is available, BitBake uses this information instead of running the main tasks.

BitBake first calls the function defined by the `BB_HASHCHECK_FUNCTION` variable with a list of tasks and corresponding hashes it wants to build. This function is designed to be fast and returns a list of the tasks for which it believes it can obtain artifacts.

Next, for each of the tasks that were returned as possibilities, BitBake executes a setscene version of the task that the possible artifact covers. Setscene versions of a task have the string "\_setscene" appended to the task name. So, for example, the task with the name `xxx` has a setscene task named `xxx_setscene`. The setscene version of the task executes and provides the necessary artifacts returning either success or failure.

As previously mentioned, an artifact can cover more than one task. For example, it is pointless to obtain a compiler if you already have the compiled binary. To handle this, BitBake calls the `BB_SETSCENE_DEPVALID` function for each successful setscene task to know whether or not it needs to obtain the dependencies of that task.

Finally, after all the setscene tasks have executed, BitBake calls the function listed in `BB_SETSCENE_VERIFY_FUNCTION2` with the list of tasks BitBake thinks has been "covered". The metadata can then ensure that this list is correct and can inform BitBake that it wants specific tasks to be run regardless of the setscene result.

You can find more information on setscene metadata in the "[Task Checksums and Setscene](#)" section.

**Chapter 43. Syntax and Operators¶****43.1. Basic Syntax**

- [43.1.1. Basic Variable Setting](#)
- [43.1.2. Modifying Existing Variables](#)
- [43.1.3. Line Joining](#)
- [43.1.4. Variable Expansion](#)
- [43.1.5. Setting a default value \(??=\)](#)
- [43.1.6. Setting a weak default value \(??=\)](#)
- [43.1.7. Immediate variable expansion \(:=\)](#)
- [43.1.8. Appending \(+=\) and prepending \(+=\) With Spaces](#)
- [43.1.9. Appending \(.=\) and Prepending \(.=\) Without Spaces](#)
- [43.1.10. Appending and Prepending \(Override Style Syntax\)](#)
- [43.1.11. Removal \(Override Style Syntax\)](#)
- [43.1.12. Override Style Operation Advantages](#)
- [43.1.13. Variable Flag Syntax](#)
- [43.1.14. Inline Python Variable Expansion](#)
- [43.1.15. Unsetting variables](#)
- [43.1.16. Providing Pathnames](#)

**43.2. Exporting Variables to the Environment****43.3. Conditional Syntax (Overrides)**

- [43.3.1. Conditional Metadata](#)
- [43.3.2. Key Expansion](#)
- [43.3.3. Examples](#)

**43.4. Sharing Functionality**

- [43.4.1. Locating Include and Class Files](#)
- [43.4.2. `inherit` Directive](#)
- [43.4.3. `include` Directive](#)
- [43.4.4. `require` Directive](#)
- [43.4.5. `INHERIT` Configuration Directive](#)

**43.5. Functions**

- [43.5.1. Shell Functions](#)
- [43.5.2. BitBake-Style Python Functions](#)
- [43.5.3. Python Functions](#)
- [43.5.4. BitBake-Style Python Functions Versus Python Functions](#)
- [43.5.5. Anonymous Python Functions](#)
- [43.5.6. Flexible Inheritance for Class Functions](#)

**43.6. Tasks**

- [43.6.1. Promoting a Function to a Task](#)
- [43.6.2. Deleting a Task](#)
- [43.6.3. Passing Information Into the Build Task Environment](#)

**43.7. Variable Flags****43.8. Events****43.9. Variants - Class Extension Mechanism****43.10. Dependencies**

- [43.10.1. Dependencies Internal to the `.bb` File](#)
- [43.10.2. Build Dependencies](#)

[43.10.3. Runtime Dependencies](#)  
[43.10.4. Recursive Dependencies](#)  
[43.10.5. Inter-Task Dependencies](#)  
[43.11. Functions You Can Call From Within Python](#)  
[43.11.1. Functions for Accessing Datastore Variables](#)  
[43.11.2. Other Functions](#)  
[43.12. Task Checksums and Setscene](#)  
[43.13. Wildcard Support in Variables](#)

Bitbake files have their own syntax. The syntax has similarities to several other languages but also has some unique features. This section describes the available syntax and operators as well as provides examples.

## 43.1. Basic Syntax¶

This section provides some basic syntax examples.

### 43.1.1. Basic Variable Setting¶

The following example sets `VARIABLE` to `"value"`. This assignment occurs immediately as the statement is parsed. It is a "hard" assignment.

```
VARIABLE = "value"
```

As expected, if you include leading or trailing spaces as part of an assignment, the spaces are retained:

```
VARIABLE = " value"
VARIABLE = "value "
```

Setting `VARIABLE` to `""` sets it to an empty string, while setting the variable to `" "` sets it to a blank space (i.e. these are not the same values).

```
VARIABLE = ""
VARIABLE = " "
```

You can use single quotes instead of double quotes when setting a variable's value. Doing so allows you to use values that contain the double quote character:

```
VARIABLE = 'I have a " in my value'
```

### Note

Unlike in Bourne shells, single quotes work identically to double quotes in all other ways. They do not suppress variable expansions.

### 43.1.2. Modifying Existing Variables¶

Sometimes you need to modify existing variables. Following are some cases where you might find you want to modify an existing variable:

- Customize a recipe that uses the variable.
- Change a variable's default value used in a `*.bbclass` file.
- Change the variable in a `*.bbappend` file to override the variable in the original recipe.
- Change the variable in a configuration file so that the value overrides an existing configuration.

Changing a variable value can sometimes depend on how the value was originally assigned and also on the desired intent of the change. In particular, when you append a value to a variable that has a default value, the resulting value might not be what you expect. In this case, the value you provide might replace the value rather than append to the default value.

If after you have changed a variable's value and something unexplained occurs, you can use BitBake to check the actual value of the suspect variable. You can make these checks for both configuration and recipe level changes:

- For configuration changes, use the following:

```
$ bitbake -e
```

This command displays variable values after the configuration files (i.e. `local.conf`, `bblayers.conf`, `bitbake.conf` and so forth) have been parsed.

### Note

Variables that are exported to the environment are preceded by the string `"export"` in the command's output.

- For recipe changes, use the following:

```
$ bitbake recipe -e | grep VARIABLE=
```

This command checks to see if the variable actually makes it into a specific recipe.

### 43.1.3. Line Joining¶

Outside of [functions](#), BitBake joins any line ending in a backslash character ("\\") with the following line before parsing statements. The most common use for the "\\ " character is to split variable assignments over multiple lines, as in the following example:

```
FOO = "bar \
 baz \
 qaz"
```

Both the "\\ " character and the newline character that follow it are removed when joining lines. Thus, no newline characters end up in the value of FOO.

Consider this additional example where the two assignments both assign "barbaz" to FOO:

```
FOO = "barbaz"

FOO = "bar\
baz"
```

#### Note

BitBake does not interpret escape sequences like "\\n" in variable values. For these to have an effect, the value must be passed to some utility that interprets escape sequences, such as `printf` or `echo -n`.

### 43.1.4. Variable Expansion¶

Variables can reference the contents of other variables using a syntax that is similar to variable expansion in Bourne shells. The following assignments result in A containing "aval" and B evaluating to "preavalpost".

```
A = "aval"
B = "pre${A}post"
```

#### Note

Unlike in Bourne shells, the curly braces are mandatory: Only `${FOO}` and not `$FOO` is recognized as an expansion of FOO.

The "=" operator does not immediately expand variable references in the right-hand side. Instead, expansion is deferred until the variable assigned to is actually used. The result depends on the current values of the referenced variables. The following example should clarify this behavior:

```
A = "${B} baz"
B = "${C} bar"
C = "foo"
At this point, ${A} equals "foo bar baz"
C = "qux"
At this point, ${A} equals "qux bar baz"
B = "norf"
At this point, ${A} equals "norf baz"
```

Contrast this behavior with the [immediate variable expansion](#) operator (i.e. ":=").

If the variable expansion syntax is used on a variable that does not exist, the string is kept as is. For example, given the following assignment, BAR expands to the literal string "\${FOO}" as long as FOO does not exist.

```
BAR = "${FOO}"
```

### 43.1.5. Setting a default value (?:)¶

You can use the "?:" operator to achieve a "softer" assignment for a variable. This type of assignment allows you to define a variable if it is undefined when the statement is parsed, but to leave the value alone if the variable has a value. Here is an example:

```
A ?= "aval"
```

If A is set at the time this statement is parsed, the variable retains its value. However, if A is not set, the variable is set to "aval".

#### Note

This assignment is immediate. Consequently, if multiple "?:" assignments to a single variable exist, the first of those ends up getting used.

### 43.1.6. Setting a weak default value (??=)¶

It is possible to use a "weaker" assignment than in the previous section by using the "??=" operator. This assignment behaves identical to "?:" except that the assignment is made at the end of the parsing process rather than immediately. Consequently, when multiple "??="

"=" assignments exist, the last one is used. Also, any "=" or "?=" assignment will override the value set with "??=". Here is an example:

```
A ??= "somevalue"
A ??= "someothervalue"
```

If `A` is set before the above statements are parsed, the variable retains its value. If `A` is not set, the variable is set to "someothervalue".

Again, this assignment is a "lazy" or "weak" assignment because it does not occur until the end of the parsing process.

### 43.1.7. Immediate variable expansion (:=)

The ":= " operator results in a variable's contents being expanded immediately, rather than when the variable is actually used:

```
T = "123"
A := "${B} ${A} test ${T}"
T = "456"
B = "${T} bval"
C = "cval"
C := "${C}append"
```

In this example, `A` contains "test 123" because `${B}` and `${A}` at the time of parsing are undefined, which leaves "test 123". And, the variable `C` contains "cvalappend" since `${C}` immediately expands to "cval".

### 43.1.8. Appending (+=) and prepending (+=) With Spaces

Appending and prepending values is common and can be accomplished using the "+=" and "=+" operators. These operators insert a space between the current value and prepended or appended value.

These operators take immediate effect during parsing. Here are some examples:

```
B = "bval"
B += "additionaldata"
C = "cval"
C += "test"
```

The variable `B` contains "bval additionaldata" and `C` contains "test cval".

### 43.1.9. Appending (.=) and Prepending (.=) Without Spaces

If you want to append or prepend values without an inserted space, use the " .= " and " .= " operators.

These operators take immediate effect during parsing. Here are some examples:

```
B = "bval"
B .= "additionaldata"
C = "cval"
C .= "test"
```

The variable `B` contains "bvaladditionaldata" and `C` contains "testcval".

### 43.1.10. Appending and Prepending (Override Style Syntax)

You can also append and prepend a variable's value using an override style syntax. When you use this syntax, no spaces are inserted.

These operators differ from the " := ", " .= ", " =.", " += ", and " =+" operators in that their effects are applied at variable expansion time rather than being immediately applied. Here are some examples:

```
B = "bval"
B_append = " additional data"
C = "cval"
C_prepend = "additional data "
D = "dval"
D_append = "additional data"
```

The variable `B` becomes "bval additional data" and `C` becomes "additional data cval". The variable `D` becomes "dvaladditional data".

## Note

You must control all spacing when you use the override syntax.

It is also possible to append and prepend to shell functions and BitBake-style Python functions. See the ["Shell Functions"](#) and ["BitBake-Style Python Functions"](#) sections for examples.

### 43.1.11. Removal (Override Style Syntax)

You can remove values from lists using the removal override style syntax. Specifying a value for removal causes all occurrences of that value to be removed from the variable.

When you use this syntax, BitBake expects one or more strings. Surrounding spaces and spacing are preserved. Here is an example:

```
F00 = "123 456 789 123456 123 456 123 456"
F00_remove = "123"
F00_remove = "456"
F002 = " abc def ghi abcdef abc def abc def def"
F002_remove = " \
def \
```

```

 abc \
 ghi \
"

```

The variable `FOO` becomes " 789 123456 " and `FOO2` becomes " jkl abcdef ".

Like `"_append"` and `"_prepend"`, `"_remove"` is applied at variable expansion time.

#### 43.1.12. Override Style Operation Advantages¶

An advantage of the override style operations `"_append"`, `"_prepend"`, and `"_remove"` as compared to the `"+="` and `"=+"` operators is that the override style operators provide guaranteed operations. For example, consider a class `foo.bbclass` that needs to add the value `"val"` to the variable `FOO`, and a recipe that uses `foo.bbclass` as follows:

```

inherit foo

FOO = "initial"

```

If `foo.bbclass` uses the `"+="` operator, as follows, then the final value of `FOO` will be `"initial"`, which is not what is desired:

```

FOO += "val"

```

If, on the other hand, `foo.bbclass` uses the `"_append"` operator, then the final value of `FOO` will be `"initial val"`, as intended:

```

FOO_append = " val"

```

#### Note

It is never necessary to use `"+="` together with `"_append"`. The following sequence of assignments appends `"barbaz"` to `FOO`:

```

FOO_append = "bar"
FOO_append = "baz"

```

The only effect of changing the second assignment in the previous example to use `"+="` would be to add a space before `"baz"` in the appended value (due to how the `"+="` operator works).

Another advantage of the override style operations is that you can combine them with other overrides as described in the ["Conditional Syntax \(Overrides\)"](#) section.

#### 43.1.13. Variable Flag Syntax¶

Variable flags are BitBake's implementation of variable properties or attributes. It is a way of tagging extra information onto a variable. You can find more out about variable flags in general in the ["Variable Flags"](#) section.

You can define, append, and prepend values to variable flags. All the standard syntax operations previously mentioned work for variable flags except for override style syntax (i.e. `"_prepend"`, `"_append"`, and `"_remove"`).

Here are some examples showing how to set variable flags:

```

FOO[a] = "abc"
FOO[b] = "123"
FOO[a] += "456"

```

The variable `FOO` has two flags: `[a]` and `[b]`. The flags are immediately set to `"abc"` and `"123"`, respectively. The `[a]` flag becomes `"abc 456"`.

No need exists to pre-define variable flags. You can simply start using them. One extremely common application is to attach some brief documentation to a BitBake variable as follows:

```

CACHE[doc] = "The directory holding the cache of the metadata."

```

#### 43.1.14. Inline Python Variable Expansion¶

You can use inline Python variable expansion to set variables. Here is an example:

```

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

```

This example results in the `DATE` variable being set to the current date.

Probably the most common use of this feature is to extract the value of variables from BitBake's internal data dictionary, `Ⓒ`. The following lines select the values of a package name and its version number, respectively:

```

PN = "${@bb.parse.BBHandler.vars_from_file(d.getVar('FILE', False),d)[0] or 'defaultpkgname'}"
PV = "${@bb.parse.BBHandler.vars_from_file(d.getVar('FILE', False),d)[1] or '1.0'}"

```

#### Note

Inline Python expressions work just like variable expansions insofar as the `"="` and `":="` operators are concerned. Given the following assignment, `Ⓕ⓪⓪()` is called each time `FOO` is expanded:

```

FOO = "${@foo()}"

```

Contrast this with the following immediate assignment, where `foo()` is only called once, while the assignment is parsed:

```
FOO := "${@foo()}"
```

For a different way to set variables with Python code during parsing, see the ["Anonymous Python Functions"](#) section.

#### 43.1.15. Unsetting variables¶

It is possible to completely remove a variable or a variable flag from BitBake's internal data dictionary by using the "unset" keyword. Here is an example:

```
unset DATE
unset do_fetch[noexec]
```

These two statements remove the `DATE` and the `do_fetch[noexec]` flag.

#### 43.1.16. Providing Pathnames¶

When specifying pathnames for use with BitBake, do not use the tilde ("`~`") character as a shortcut for your home directory. Doing so might cause BitBake to not recognize the path since BitBake does not expand this character in the same way a shell would.

Instead, provide a fuller path as the following example illustrates:

```
BBLAYERS ?= " \
/home/scott-lenovo/LayerA \
"
```

### 43.2. Exporting Variables to the Environment¶

You can export variables to the environment of running tasks by using the `export` keyword. For example, in the following example, the `do_foo` task prints "value from the environment" when run:

```
export ENV_VARIABLE
ENV_VARIABLE = "value from the environment"

do_foo() {
 bbplain "$ENV_VARIABLE"
}
```

#### Note

BitBake does not expand `$ENV_VARIABLE` in this case because it lacks the obligatory `{ }`. Rather, `$ENV_VARIABLE` is expanded by the shell.

It does not matter whether `export ENV_VARIABLE` appears before or after assignments to `ENV_VARIABLE`.

It is also possible to combine `export` with setting a value for the variable. Here is an example:

```
export ENV_VARIABLE = "variable-value"
```

In the output of `bitbake -e`, variables that are exported to the environment are preceded by "export".

Among the variables commonly exported to the environment are `CC` and `CFLAGS`, which are picked up by many build systems.

### 43.3. Conditional Syntax (Overrides)¶

BitBake uses [OVERRIDES](#) to control what variables are overridden after BitBake parses recipes and configuration files. This section describes how you can use [OVERRIDES](#) as conditional metadata, talks about key expansion in relationship to [OVERRIDES](#), and provides some examples to help with understanding.

#### 43.3.1. Conditional Metadata¶

You can use [OVERRIDES](#) to conditionally select a specific version of a variable and to conditionally append or prepend the value of a variable.

#### Note

Overrides can only use lower-case characters. Additionally, underscores are not permitted in override names as they are used to separate overrides from each other and from the variable name.

- **Selecting a Variable:** The [OVERRIDES](#) variable is a colon-character-separated list that contains items for which you want to satisfy conditions. Thus, if you have a variable that is conditional on "arm", and "arm" is in [OVERRIDES](#), then the "arm"-specific version of the variable is used rather than the non-conditional version. Here is an example:

```

OVERRIDES = "architecture:os:machine"
TEST = "default"
TEST_os = "ospecific"
TEST_nooverride = "othercondvalue"

```

In this example, the `OVERRIDES` variable lists three overrides: "architecture", "os", and "machine". The variable `TEST` by itself has a default value of "default". You select the os-specific version of the `TEST` variable by appending the "os" override to the variable (i.e. `TEST_os`).

To better understand this, consider a practical example that assumes an OpenEmbedded metadata-based Linux kernel recipe file. The following lines from the recipe file first set the kernel branch variable `KBRANCH` to a default value, then conditionally override that value based on the architecture of the build:

```

KBRANCH = "standard/base"
KBRANCH_qemuarm = "standard/arm-versatile-926ejs"
KBRANCH_qemumips = "standard/mti-malta32"
KBRANCH_qemuppc = "standard/qemuppc"
KBRANCH_qemux86 = "standard/common-pc/base"
KBRANCH_qemux86-64 = "standard/common-pc-64/base"
KBRANCH_qemumips64 = "standard/mti-malta64"

```

- **Appending and Prepending:** BitBake also supports append and prepend operations to variable values based on whether a specific item is listed in `OVERRIDES`. Here is an example:

```

DEPENDS = "glibc ncurses"
OVERRIDES = "machine:local"
DEPENDS_append_machine = " libmad"

```

In this example, `DEPENDS` becomes "glibc ncurses libmad".

Again, using an OpenEmbedded metadata-based kernel recipe file as an example, the following lines will conditionally append to the `KERNEL_FEATURES` variable based on the architecture:

```

KERNEL_FEATURES_append = "${KERNEL_EXTRA_FEATURES}"
KERNEL_FEATURES_append_qemux86=" cfg/sound.scc cfg/paravirt_kvm.scc"
KERNEL_FEATURES_append_qemux86-64=" cfg/sound.scc cfg/paravirt_kvm.scc"

```

- **Setting a Variable for a Single Task:** BitBake supports setting a variable just for the duration of a single task. Here is an example:

```

FOO_task-configure = "val 1"
FOO_task-compile = "val 2"

```

In the previous example, `FOO` has the value "val 1" while the `do_configure` task is executed, and the value "val 2" while the `do_compile` task is executed.

Internally, this is implemented by prepending the task (e.g. "task-compile:") to the value of `OVERRIDES` for the local datastore of the `do_compile` task.

You can also use this syntax with other combinations (e.g. "`__prepend`") as shown in the following example:

```

EXTRA_OEMAKE_prepend_task-compile = "${PARALLEL_MAKE} "

```

### 43.3.2. Key Expansion¶

Key expansion happens when the BitBake datastore is finalized. To better understand this, consider the following example:

```

A${B} = "X"
B = "2"
A2 = "Y"

```

In this case, after all the parsing is complete, BitBake expands `${B}` into "2". This expansion causes `A2`, which was set to "Y" before the expansion, to become "X".

### 43.3.3. Examples¶

Despite the previous explanations that show the different forms of variable definitions, it can be hard to work out exactly what happens when variable operators, conditional overrides, and unconditional overrides are combined. This section presents some common scenarios along with explanations for variable interactions that typically confuse users.

There is often confusion concerning the order in which overrides and various "append" operators take effect. Recall that an append or prepend operation using "`__append`" and "`__prepend`" does not result in an immediate assignment as would "+=", "=", "+=", or "=". Consider the following example:

```

OVERRIDES = "foo"
A = "Z"
A_foo_append = "X"

```

For this case, `A` is unconditionally set to "Z" and "X" is unconditionally and immediately appended to the variable `A_foo`. Because overrides have not been applied yet, `A_foo` is set to "X" due to the append and `A` simply equals "Z".

Applying overrides, however, changes things. Since "foo" is listed in `OVERRIDES`, the conditional variable `A` is replaced with the "foo" version, which is equal to "X". So effectively, `A_foo` replaces `A`.

This next example changes the order of the override and the append:

```

OVERRIDES = "foo"
A = "Z"
A_append_foo = "X"

```



For this case, before overrides are handled, `A` is set to "Z" and `A__append_foo` is set to "X". Once the override for "foo" is applied, however, `A` gets appended with "X". Consequently, `A` becomes "ZX". Notice that spaces are not appended.

This next example has the order of the appends and overrides reversed back as in the first example:

```

OVERRIDES = "foo"
A = "Y"
A__foo_append = "Z"
A__foo_append = "X"

```

For this case, before any overrides are resolved, `A` is set to "Y" using an immediate assignment. After this immediate assignment, `A__foo` is set to "Z", and then further appended with "X" leaving the variable set to "ZX". Finally, applying the override for "foo" results in the conditional variable `A` becoming "ZX" (i.e. `A` is replaced with `A__foo`).

This final example mixes in some varying operators:

```

A = "1"
A__append = "2"
A__append = "3"
A += "4"
A .= "5"

```

For this case, the type of append operators are affecting the order of assignments as BitBake passes through the code multiple times. Initially, `A` is set to "1 45" because of the three statements that use immediate operators. After these assignments are made, BitBake applies the "\_append" operations. Those operations result in `A` becoming "1 4523".

## 43.4. Sharing Functionality¶

BitBake allows for metadata sharing through include files (`.inc`) and class files (`.bbclass`). For example, suppose you have a piece of common functionality such as a task definition that you want to share between more than one recipe. In this case, creating a `.bbclass` file that contains the common functionality and then using the `inherit` directive in your recipes to inherit the class would be a common way to share the task.

This section presents the mechanisms BitBake provides to allow you to share functionality between recipes. Specifically, the mechanisms include `include`, `inherit`, `INHERIT`, and `require` directives.

### 43.4.1. Locating Include and Class Files¶

BitBake uses the `BBPATH` variable to locate needed include and class files. Additionally, BitBake searches the current directory for `include` and `require` directives.

#### Note

The `BBPATH` variable is analogous to the environment variable `PATH`.

In order for `include` and class files to be found by BitBake, they need to be located in a "classes" subdirectory that can be found in `BBPATH`.

### 43.4.2. inherit Directive¶

When writing a recipe or class file, you can use the `inherit` directive to inherit the functionality of a class (`.bbclass`). BitBake only supports this directive when used within recipe and class files (i.e. `.bb` and `.bbclass`).

The `inherit` directive is a rudimentary means of specifying functionality contained in class files that your recipes require. For example, you can easily abstract out the tasks involved in building a package that uses Autoconf and Automake and put those tasks into a class file and then have your recipe inherit that class file.

As an example, your recipes could use the following directive to inherit an `autotools.bbclass` file. The class file would contain common functionality for using Autotools that could be shared across recipes:

```
inherit autotools
```

In this case, BitBake would search for the directory `classes/autotools.bbclass` in `BBPATH`.

#### Note

You can override any values and functions of the inherited class within your recipe by doing so after the "inherit" statement.

If you want to use the directive to inherit multiple classes, separate them with spaces. The following example shows how to inherit both the `buildhistory` and `rm_work` classes:

```
inherit buildhistory rm_work
```

An advantage with the `inherit` directive as compared to both the `include` and `require` directives is that you can inherit class files conditionally. You can accomplish this by using a variable expression after the `inherit` statement. Here is an example:

```
inherit ${VARNAME}
```

If `VARNAME` is going to be set, it needs to be set before the `inherit` statement is parsed. One way to achieve a conditional `inherit` in this case is to use overrides:

```
VARIABLE = ""
VARIABLE_someoverride = "myclass"
```

Another method is by using anonymous Python. Here is an example:

```
python () {
 if condition == value:
 d.setVar('VARIABLE', 'myclass')
 else:
 d.setVar('VARIABLE', '')
}
```

Alternatively, you could use an in-line Python expression in the following form:

```
inherit ${@'classname' if condition else ''}
inherit ${@functionname(params)}
```

In all cases, if the expression evaluates to an empty string, the statement does not trigger a syntax error because it becomes a no-op.

#### 43.4.3. `include` Directive¶

BitBake understands the `include` directive. This directive causes BitBake to parse whatever file you specify, and to insert that file at that location. The directive is much like its equivalent in Make except that if the path specified on the include line is a relative path, BitBake locates the first file it can find within `BBPATH`.

The `include` directive is a more generic method of including functionality as compared to the `inherit` directive, which is restricted to class (i.e. `.bbclass`) files. The `include` directive is applicable for any other kind of shared or encapsulated functionality or configuration that does not suit a `.bbclass` file.

As an example, suppose you needed a recipe to include some self-test definitions:

```
include test_defs.inc
```

#### Note

The `include` directive does not produce an error when the file cannot be found. Consequently, it is recommended that if the file you are including is expected to exist, you should use `require` instead of `include`. Doing so makes sure that an error is produced if the file cannot be found.

#### 43.4.4. `require` Directive¶

BitBake understands the `require` directive. This directive behaves just like the `include` directive with the exception that BitBake raises a parsing error if the file to be included cannot be found. Thus, any file you `require` is inserted into the file that is being parsed at the location of the directive.

The `require` directive, like the `include` directive previously described, is a more generic method of including functionality as compared to the `inherit` directive, which is restricted to class (i.e. `.bbclass`) files. The `require` directive is applicable for any other kind of shared or encapsulated functionality or configuration that does not suit a `.bbclass` file.

Similar to how BitBake handles `include`, if the path specified on the `require` line is a relative path, BitBake locates the first file it can find within `BBPATH`.

As an example, suppose you have two versions of a recipe (e.g. `foo_1.2.2.bb` and `foo_2.0.0.bb`) where each version contains some identical functionality that could be shared. You could create an include file named `foo.inc` that contains the common definitions needed to build "foo". You need to be sure `foo.inc` is located in the same directory as your two recipe files as well. Once these conditions are set up, you can share the functionality using a `require` directive from within each recipe:

```
require foo.inc
```

#### 43.4.5. `INHERIT` Configuration Directive¶

When creating a configuration file (`.conf`), you can use the `INHERIT` configuration directive to inherit a class. BitBake only supports this directive when used within a configuration file.

As an example, suppose you needed to inherit a class file called `abc.bbclass` from a configuration file as follows:

```
INHERIT += "abc"
```

This configuration directive causes the named class to be inherited at the point of the directive during parsing. As with the `inherit` directive, the `.bbclass` file must be located in a "classes" subdirectory in one of the directories specified in `BBPATH`.

#### Note

Because `.conf` files are parsed first during BitBake's execution, using `INHERIT` to inherit a class effectively inherits the class globally (i.e. for all recipes).

If you want to use the directive to inherit multiple classes, you can provide them on the same line in the `local.conf` file. Use spaces to separate the classes. The following example shows how to inherit both the `autotools` and `pkgconfig` classes:

```
INHERIT += "autotools pkgconfig"
```

## 43.5. Functions¶

As with most languages, functions are the building blocks that are used to build up operations into tasks. BitBake supports these types of functions:

- **Shell Functions:** Functions written in shell script and executed either directly as functions, tasks, or both. They can also be called by other shell functions.
- **BitBake-Style Python Functions:** Functions written in Python and executed by BitBake or other Python functions using `bb.build.exec_func()`.
- **Python Functions:** Functions written in Python and executed by Python.
- **Anonymous Python Functions:** Python functions executed automatically during parsing.

Regardless of the type of function, you can only define them in class (`.bbclass`) and recipe (`.bb` or `.inc`) files.

### 43.5.1. Shell Functions¶

Functions written in shell script and executed either directly as functions, tasks, or both. They can also be called by other shell functions. Here is an example shell function definition:

```
some_function () {
 echo "Hello World"
}
```

When you create these types of functions in your recipe or class files, you need to follow the shell programming rules. The scripts are executed by `/bin/sh`, which may not be a bash shell but might be something such as `dash`. You should not use Bash-specific script (bashisms).

Overrides and override-style operators like `_append` and `_prepend` can also be applied to shell functions. Most commonly, this application would be used in a `.bbappend` file to modify functions in the main recipe. It can also be used to modify functions inherited from classes.

As an example, consider the following:

```
do_foo() {
 bbplain first
 fn
}

fn_prepend() {
 bbplain second
}

fn() {
 bbplain third
}

do_foo_append() {
 bbplain fourth
}
```

Running `do_foo` prints the following:

```
recipe: do_foo: first
recipe: do_foo: second
recipe: do_foo: third
recipe: do_foo: fourth
```

### Note

Overrides and override-style operators can be applied to any shell function, not just tasks.

You can use the `bitbake -e recipe:do_foo` command to view the final assembled function after all overrides have been applied.

### 43.5.2. BitBake-Style Python Functions¶

These functions are written in Python and executed by BitBake or other Python functions using `bb.build.exec_func()`.

An example BitBake function is:

```
python some_python_function () {
 d.setVar("TEXT", "Hello World")
 print d.getVar("TEXT")
}
```

Because the Python "bb" and "os" modules are already imported, you do not need to import these modules. Also in these types of functions, the datastore ("d") is a global variable and is always automatically available.

## Note

Variable expressions (e.g. `$(X)`) are no longer expanded within Python functions. This behavior is intentional in order to allow you to freely set variable values to expandable expressions without having them expanded prematurely. If you do wish to expand a variable within a Python function, use `d.getVar("X")`. Or, for more complicated expressions, use `d.expand()`.

Similar to shell functions, you can also apply overrides and override-style operators to BitBake-style Python functions.

As an example, consider the following:

```
python do_foo_prepend() {
 bb.plain("first")
}

python do_foo() {
 bb.plain("second")
}

python do_foo_append() {
 bb.plain("third")
}
```

Running `do_foo` prints the following:

```
recipeName do_foo: first
recipeName do_foo: second
recipeName do_foo: third
```

You can use the `bitbake -e recipeName` command to view the final assembled function after all overrides have been applied.

### 43.5.3. Python Functions¶

These functions are written in Python and are executed by other Python code. Examples of Python functions are utility functions that you intend to call from in-line Python or from within other Python functions. Here is an example:

```
def get_depends(d):
 if d.getVar('SOMECONDITION'):
 return "dependencywithcond"
 else:
 return "dependency"
SOMECONDITION = "1"
DEPENDS = "${@get_depends(d)}"
```

This would result in `DEPENDS` containing `dependencywithcond`.

Here are some things to know about Python functions:

- Python functions can take parameters.
- The BitBake datastore is not automatically available. Consequently, you must pass it in as a parameter to the function.
- The "bb" and "os" Python modules are automatically available. You do not need to import them.

### 43.5.4. Bitbake-Style Python Functions Versus Python Functions¶

Following are some important differences between BitBake-style Python functions and regular Python functions defined with "def":

- Only BitBake-style Python functions can be [tasks](#).
- Overrides and override-style operators can only be applied to BitBake-style Python functions.
- Only regular Python functions can take arguments and return values.
- [Variable flags](#) such as `[dirs]`, `[cleandirs]`, and `[lockfiles]` can be used on BitBake-style Python functions, but not on regular Python functions.
- BitBake-style Python functions generate a separate `$(T)/run.function-name.pid` script that is executed to run the function, and also generate a log file in `$(T)/log.function-name.pid` if they are executed as tasks.

Regular Python functions execute "inline" and do not generate any files in `$(T)`.

- Regular Python functions are called with the usual Python syntax. BitBake-style Python functions are usually tasks and are called directly by BitBake, but can also be called manually from Python code by using the `bb.build.exec_func()` function. Here is an example:

```
bb.build.exec_func("my_bitbake_style_function", d)
```

## Note

`bb.build.exec_func()` can also be used to run shell functions from Python code. If you want to run a shell function before a Python function within the same task, then you can use a parent helper Python function that starts by running the shell function with `bb.build.exec_func()` and then runs the Python code.

To detect errors from functions executed with `bb.build.exec_func()`, you can catch the `bb.build.FuncFailed` exception.

## Note

Functions in metadata (recipes and classes) should not themselves raise `bb.build.FuncFailed`. Rather, `bb.build.FuncFailed` should be viewed as a general indicator that the called function failed by raising an exception. For example, an exception raised by `bb.fatal()` will be caught inside `bb.build.exec_func()`, and a `bb.build.FuncFailed` will be raised in response.

Due to their simplicity, you should prefer regular Python functions over BitBake-style Python functions unless you need a feature specific to BitBake-style Python functions. Regular Python functions in metadata are a more recent invention than BitBake-style Python functions, and older code tends to use `bb.build.exec_func()` more often.

### 43.5.5. Anonymous Python Functions¶

Sometimes it is useful to set variables or perform other operations programmatically during parsing. To do this, you can define special Python functions, called anonymous Python functions, that run at the end of parsing. For example, the following conditionally sets a variable based on the value of another variable:

```
python () {
 if d.getVar('SOMEVAR') == 'value':
 d.setVar('ANOTHERVAR', 'value2')
}
```

An equivalent way to mark a function as an anonymous function is to give it the name `"__anonymous"`, rather than no name.

Anonymous Python functions always run at the end of parsing, regardless of where they are defined. If a recipe contains many anonymous functions, they run in the same order as they are defined within the recipe. As an example, consider the following snippet:

```
python () {
 d.setVar('FOO', 'foo 2')
}

FOO = "foo 1"

python () {
 d.appendVar('BAR', ' bar 2')
}

BAR = "bar 1"
```

The previous example is conceptually equivalent to the following snippet:

```
FOO = "foo 1"
BAR = "bar 1"
FOO = "foo 2"
BAR += "bar 2"
```

`FOO` ends up with the value `"foo 2"`, and `BAR` with the value `"bar 1 bar 2"`. Just as in the second snippet, the values set for the variables within the anonymous functions become available to tasks, which always run after parsing.

Overrides and override-style operators such as `"_append"` are applied before anonymous functions run. In the following example, `FOO` ends up with the value `"foo from anonymous"`:

```
FOO = "foo"
FOO_append = " from outside"

python () {
 d.setVar("FOO", "foo from anonymous")
}
```

For methods you can use with anonymous Python functions, see the ["Functions You Can Call From Within Python"](#) section. For a different method to run Python code during parsing, see the ["Inline Python Variable Expansion"](#) section.

### 43.5.6. Flexible Inheritance for Class Functions¶

Through coding techniques and the use of `EXPORT_FUNCTIONS`, BitBake supports exporting a function from a class such that the class function appears as the default implementation of the function, but can still be called if a recipe inheriting the class needs to define its own version of the function.

To understand the benefits of this feature, consider the basic scenario where a class defines a task function and your recipe inherits the class. In this basic scenario, your recipe inherits the task function as defined in the class. If desired, your recipe can add to the start and end of the function by using the `"_prepend"` or `"_append"` operations respectively, or it can redefine the function completely. However, if it redefines the function, there is no means for it to call the class version of the function. `EXPORT_FUNCTIONS` provides a mechanism that enables the recipe's version of the function to call the original version of the function.

To make use of this technique, you need the following things in place:

- The class needs to define the function as follows:

```
classname_functionname
```

For example, if you have a class file `bar.bbclass` and a function named `do_foo`, the class must define the function as follows:

```
bar_do_foo
```

- The class needs to contain the `EXPORT_FUNCTIONS` statement as follows:

```
EXPORT_FUNCTIONS functionname
```

For example, continuing with the same example, the statement in the `bar.bbclass` would be as follows:

```
EXPORT_FUNCTIONS do_foo
```

- You need to call the function appropriately from within your recipe. Continuing with the same example, if your recipe needs to call the class version of the function, it should call `bar_do_foo`. Assuming `do_foo` was a shell function and `EXPORT_FUNCTIONS` was used as above, the recipe's function could conditionally call the class version of the function as follows:

```
do_foo() {
 if [somecondition] ; then
 bar_do_foo
 else
 # Do something else
 fi
}
```

To call your modified version of the function as defined in your recipe, call it as `do_foo`.

With these conditions met, your single recipe can freely choose between the original function as defined in the class file and the modified function in your recipe. If you do not set up these conditions, you are limited to using one function or the other.

## 43.6. Tasks¶

Tasks are BitBake execution units that make up the steps that BitBake can run for a given recipe. Tasks are only supported in recipes and classes (i.e. in `.bb` files and files included or inherited from `.bb` files). By convention, tasks have names that start with "do\_".

### 43.6.1. Promoting a Function to a Task¶

Tasks are either [shell functions](#) or [BitBake-style Python functions](#) that have been promoted to tasks by using the `addtask` command. The `addtask` command can also optionally describe dependencies between the task and other tasks. Here is an example that shows how to define a task and declare some dependencies:

```
python do_printdate () {
 import time
 print time.strftime('%Y%m%d', time.gmtime())
}
addtask printdate after do_fetch before do_build
```

The first argument to `addtask` is the name of the function to promote to a task. If the name does not start with "do\_", "do\_" is implicitly added, which enforces the convention that all task names start with "do\_".

In the previous example, the `do_printdate` task becomes a dependency of the `do_build` task, which is the default task (i.e. the task run by the `bitbake` command unless another task is specified explicitly). Additionally, the `do_printdate` task becomes dependent upon the `do_fetch` task. Running the `do_build` task results in the `do_printdate` task running first.

## Note

If you try out the previous example, you might see that the `do_printdate` task is only run the first time you build the recipe with the `bitbake` command. This is because BitBake considers the task "up-to-date" after that initial run. If you want to force the task to always be rerun for experimentation purposes, you can make BitBake always consider the task "out-of-date" by using the `[nostamp]` variable flag, as follows:

```
do_printdate[nostamp] = "1"
```

You can also explicitly run the task and provide the `-f` option as follows:

```
$ bitbake recipe -c printdate -f
```

When manually selecting a task to run with the `bitbake recipe -c task` command, you can omit the "do\_" prefix as part of the task name.

You might wonder about the practical effects of using `addtask` without specifying any dependencies as is done in the following example:

```
addtask printdate
```

In this example, assuming dependencies have not been added through some other means, the only way to run the task is by explicitly selecting it with `bitbake recipe -c printdate`. You can use the `do_listtasks` task to list all tasks defined in a recipe as shown in the following example:

```
$ bitbake recipe -c listtasks
```

For more information on task dependencies, see the ["Dependencies"](#) section.

See the ["Variable Flags"](#) section for information on variable flags you can use with tasks.

### 43.6.2. Deleting a Task¶

As well as being able to add tasks, you can delete them. Simply use the `deltask` command to delete a task. For example, to delete the example task used in the previous sections, you would use:

```
deltask printdate
```

If you delete a task using the `deltask` command and the task has dependencies, the dependencies are not reconnected. For example, suppose you have three tasks named `do_a`, `do_b`, and `do_c`. Furthermore, `do_c` is dependent on `do_b`, which in turn is dependent on `do_a`. Given this scenario, if you use `deltask` to delete `do_b`, the implicit dependency relationship between `do_c` and `do_a` through `do_b` no longer exists, and `do_c` dependencies are not updated to include `do_a`. Thus, `do_c` is free to run before `do_a`.

If you want dependencies such as these to remain intact, use the `[noexec]` varflag to disable the task instead of using the `deltask` command to delete it:

```
do_b[noexec] = "1"
```

### 43.6.3. Passing Information Into the Build Task Environment¶

When running a task, BitBake tightly controls the shell execution environment of the build tasks to make sure unwanted contamination from the build machine cannot influence the build.

#### Note

By default, BitBake cleans the environment to include only those things exported or listed in its whitelist to ensure that the build environment is reproducible and consistent. You can prevent this "cleaning" by setting the `BB_PRESERVE_ENV` variable.

Consequently, if you do want something to get passed into the build task environment, you must take these two steps:

1. Tell BitBake to load what you want from the environment into the datastore. You can do so through the `BB_ENV_WHITELIST` and `BB_ENV_EXTRAWHITE` variables. For example, assume you want to prevent the build system from accessing your `$HOME/.ccache` directory. The following command "whitelists" the environment variable `CCACHE_DIR` causing BitBake to allow that variable into the datastore:

```
export BB_ENV_EXTRAWHITE="$BB_ENV_EXTRAWHITE CCACHE_DIR"
```

2. Tell BitBake to export what you have loaded into the datastore to the task environment of every running task. Loading something from the environment into the datastore (previous step) only makes it available in the datastore. To export it to the task environment of every running task, use a command similar to the following in your local configuration file `local.conf` or your distribution configuration file:

```
export CCACHE_DIR
```

#### Note

A side effect of the previous steps is that BitBake records the variable as a dependency of the build process in things like the setscene checksums. If doing so results in unnecessary rebuilds of tasks, you can whitelist the variable so that the setscene code ignores the dependency when it creates checksums.

Sometimes, it is useful to be able to obtain information from the original execution environment. BitBake saves a copy of the original environment into a special variable named `BB_ORIGENV`.

The `BB_ORIGENV` variable returns a datastore object that can be queried using the standard datastore operators such as `getVar(, False)`. The datastore object is useful, for example, to find the original `DISPLAY` variable. Here is an example:

```
origenv = d.getVar("BB_ORIGENV", False)
bar = origenv.getVar("BAR", False)
```

The previous example returns `BAR` from the original execution environment.

## 43.7. Variable Flags¶

Variable flags (varflags) help control a task's functionality and dependencies. BitBake reads and writes varflags to the datastore using the following command forms:

```
variable = d.getVarFlags("variable")
self.d.setVarFlags("FOO", {"func": True})
```

When working with varflags, the same syntax, with the exception of overrides, applies. In other words, you can set, append, and prepend varflags just like variables. See the ["Variable Flag Syntax"](#) section for details.

BitBake has a defined set of varflags available for recipes and classes. Tasks support a number of these flags which control various functionality of the task:

- **[cleandirs]:** Empty directories that should be created before the task runs. Directories that already exist are removed and recreated to empty them.

- **[depends]**: Controls inter-task dependencies. See the [DEPENDS](#) variable and the "Inter-Task Dependencies" section for more information.
- **[deptask]**: Controls task build-time dependencies. See the [DEPENDS](#) variable and the "Build Dependencies" section for more information.
- **[dirs]**: Directories that should be created before the task runs. Directories that already exist are left as is. The last directory listed is used as the current working directory for the task.
- **[lockfiles]**: Specifies one or more lockfiles to lock while the task executes. Only one task may hold a lockfile, and any task that attempts to lock an already locked file will block until the lock is released. You can use this variable flag to accomplish mutual exclusion.
- **[noexec]**: When set to "1", marks the task as being empty, with no execution required. You can use the `[noexec]` flag to set up tasks as dependency placeholders, or to disable tasks defined elsewhere that are not needed in a particular recipe.
- **[nostamp]**: When set to "1", tells BitBake to not generate a stamp file for a task, which implies the task should always be executed.

## Caution

Any task that depends (possibly indirectly) on a `[nostamp]` task will always be executed as well. This can cause unnecessary rebuilding if you are not careful.

- **[number\_threads]**: Limits tasks to a specific number of simultaneous threads during execution. This varflag is useful when your build host has a large number of cores but certain tasks need to be rate-limited due to various kinds of resource constraints (e.g. to avoid network throttling). `number_threads` works similarly to the [BB\\_NUMBER\\_THREADS](#) variable but is task-specific.

Set the value globally. For example, the following makes sure the `do_fetch` task uses no more than two simultaneous execution threads:

```
do_fetch[number_threads] = "2"
```

## Warnings

- Setting the varflag in individual recipes rather than globally can result in unpredictable behavior.
- Setting the varflag to a value greater than the value used in the [BB\\_NUMBER\\_THREADS](#) variable causes `number_threads` to have no effect.

- **[postfuncs]**: List of functions to call after the completion of the task.
- **[prefuncs]**: List of functions to call before the task executes.
- **[rdepends]**: Controls inter-task runtime dependencies. See the [RDEPENDS](#) variable, the [RRECOMMENDS](#) variable, and the "Inter-Task Dependencies" section for more information.
- **[rdeptask]**: Controls task runtime dependencies. See the [RDEPENDS](#) variable, the [RRECOMMENDS](#) variable, and the "Runtime Dependencies" section for more information.
- **[recideptask]**: When set in conjunction with `recrdeptask`, specifies a task that should be inspected for additional dependencies.
- **[recrdeptask]**: Controls task recursive runtime dependencies. See the [RDEPENDS](#) variable, the [RRECOMMENDS](#) variable, and the "Recursive Dependencies" section for more information.
- **[stamp-extra-info]**: Extra stamp information to append to the task's stamp. As an example, OpenEmbedded uses this flag to allow machine-specific tasks.
- **[umask]**: The umask to run the task under.

Several varflags are useful for controlling how signatures are calculated for variables. For more information on this process, see the "Checksums (Signatures)" section.

- **[vardeps]**: Specifies a space-separated list of additional variables to add to a variable's dependencies for the purposes of calculating its signature. Adding variables to this list is useful, for example, when a function refers to a variable in a manner that does not allow BitBake to automatically determine that the variable is referred to.
- **[vardepsexclude]**: Specifies a space-separated list of variables that should be excluded from a variable's dependencies for the purposes of calculating its signature.
- **[vardepvalue]**: If set, instructs BitBake to ignore the actual value of the variable and instead use the specified value when calculating the variable's signature.
- **[vardepvalueexclude]**: Specifies a pipe-separated list of strings to exclude from the variable's value when calculating the variable's signature.

## 43.8. Events ¶

BitBake allows installation of event handlers within recipe and class files. Events are triggered at certain points during operation, such as the beginning of operation against a given recipe (i.e. `* .bb`), the start of a given task, a task failure, a task success, and so forth. The intent is to make it easy to do things like email notification on build failures.



Following is an example event handler that prints the name of the event and the content of the `FILE` variable:

```
addhandler myclass_eventhandler
python myclass_eventhandler() {
 from bb.event import getName
 print("The name of the Event is %s" % getName(e))
 print("The file we run for is %s" % d.getVar('FILE'))
}
myclass_eventhandler[eventmask] = "bb.event.BuildStarted bb.event.BuildCompleted"
```

In the previous example, an eventmask has been set so that the handler only sees the "BuildStarted" and "BuildCompleted" events. This event handler gets called every time an event matching the eventmask is triggered. A global variable "e" is defined, which represents the current event. With the `getName(e)` method, you can get the name of the triggered event. The global datastore is available as "d". In legacy code, you might see "e.data" used to get the datastore. However, realize that "e.data" is deprecated and you should use "d" going forward.

The context of the datastore is appropriate to the event in question. For example, "BuildStarted" and "BuildCompleted" events run before any tasks are executed so would be in the global configuration datastore namespace. No recipe-specific metadata exists in that namespace. The "BuildStarted" and "BuildCompleted" events also run in the main cooker/server process rather than any worker context. Thus, any changes made to the datastore would be seen by other cooker/server events within the current build but not seen outside of that build or in any worker context. Task events run in the actual tasks in question consequently have recipe-specific and task-specific contents. These events run in the worker context and are discarded at the end of task execution.

During a standard build, the following common events might occur. The following events are the most common kinds of events that most metadata might have an interest in viewing:

- `bb.event.ConfigParsed()`: Fired when the base configuration; which consists of `bitbake.conf`, `base.bbclass` and any global `INHERIT` statements; has been parsed. You can see multiple such events when each of the workers parse the base configuration or if the server changes configuration and reparses. Any given datastore only has one such event executed against it, however. If `BB_INVALIDCONF` is set in the datastore by the event handler, the configuration is reparsed and a new event triggered, allowing the metadata to update configuration.
- `bb.event.HeartbeatEvent()`: Fires at regular time intervals of one second. You can configure the interval time using the `BB_HEARTBEAT_EVENT` variable. The event's "time" attribute is the `time.time()` value when the event is triggered. This event is useful for activities such as system state monitoring.
- `bb.event.ParseStarted()`: Fired when BitBake is about to start parsing recipes. This event's "total" attribute represents the number of recipes BitBake plans to parse.
- `bb.event.ParseProgress()`: Fired as parsing progresses. This event's "current" attribute is the number of recipes parsed as well as the "total" attribute.
- `bb.event.ParseCompleted()`: Fired when parsing is complete. This event's "cached", "parsed", "skipped", "virtuals", "masked", and "errors" attributes provide statistics for the parsing results.
- `bb.event.BuildStarted()`: Fired when a new build starts. BitBake fires multiple "BuildStarted" events (one per configuration) when multiple configuration (multiconfig) is enabled.
- `bb.build.TaskStarted()`: Fired when a task starts. This event's "taskfile" attribute points to the recipe from which the task originates. The "taskname" attribute, which is the task's name, includes the `do_` prefix, and the "logfile" attribute point to where the task's output is stored. Finally, the "time" attribute is the task's execution start time.
- `bb.build.TaskInvalid()`: Fired if BitBake tries to execute a task that does not exist.
- `bb.build.TaskFailedSilent()`: Fired for setscene tasks that fail and should not be presented to the user verbosely.
- `bb.build.TaskFailed()`: Fired for normal tasks that fail.
- `bb.build.TaskSucceeded()`: Fired when a task successfully completes.
- `bb.event.BuildCompleted()`: Fired when a build finishes.
- `bb.cooker.CookerExit()`: Fired when the BitBake server/cooker shuts down. This event is usually only seen by the UIs as a sign they should also shutdown.

This next list of example events occur based on specific requests to the server. These events are often used to communicate larger pieces of information from the BitBake server to other parts of BitBake such as user interfaces:

- `bb.event.TreeDataPreparationStarted()`
- `bb.event.TreeDataPreparationProgress()`
- `bb.event.TreeDataPreparationCompleted()`
- `bb.event.DepTreeGenerated()`
- `bb.event.CoreBaseFilesFound()`
- `bb.event.ConfigFilePathFound()`
- `bb.event.FilesMatchingFound()`
- `bb.event.ConfigFilesFound()`
- `bb.event.TargetsTreeGenerated()`

## 43.9. Variants - Class Extension Mechanism ¶

BitBake supports two features that facilitate creating from a single recipe file multiple incarnations of that recipe file where all incarnations are buildable. These features are enabled through the `BBCLASSEXTEND` and `BBVERSIONS` variables.

## Note

The mechanism for this class extension is extremely specific to the implementation. Usually, the recipe's `PROVIDES`, `PN`, and `DEPENDS` variables would need to be modified by the extension class. For specific examples, see the OE-Core `native`, `nativesdk`, and `multilib` classes.

- **BBCLASSEXTEND:** This variable is a space separated list of classes used to "extend" the recipe for each variant. Here is an example that results in a second incarnation of the current recipe being available. This second incarnation will have the "native" class inherited.

```
BBCLASSEXTEND = "native"
```

- **BBVERSIONS:** This variable allows a single recipe to build multiple versions of a project from a single recipe file. You can also specify conditional metadata (using the `OVERRIDES` mechanism) for a single version, or an optionally named range of versions. Here is an example:

```
BBVERSIONS = "1.0 2.0 git"
SRC_URI_git = "git://someurl/somepath.git"

BBVERSIONS = "1.0.[0-6]:1.0.0+ \ 1.0.[7-9]:1.0.7+"
SRC_URI_append_1.0.7+ = "file://some_patch_which_the_new_versions_need.patch;patch=1"
```

The name of the range defaults to the original version of the recipe. For example, in OpenEmbedded, the recipe file `foo_1.0.0+.bb` creates a default name range of `1.0.0+`. This is useful because the range name is not only placed into overrides, but it is also made available for the metadata to use in the variable that defines the base recipe versions for use in `file://` search paths (`FILESPATH`).

## 43.10. Dependencies¶

To allow for efficient parallel processing, BitBake handles dependencies at the task level. Dependencies can exist both between tasks within a single recipe and between tasks in different recipes. Following are examples of each:

- For tasks within a single recipe, a recipe's `do_configure` task might need to complete before its `do_compile` task can run.
- For tasks in different recipes, one recipe's `do_configure` task might require another recipe's `do_populate_sysroot` task to finish first such that the libraries and headers provided by the other recipe are available.

This section describes several ways to declare dependencies. Remember, even though dependencies are declared in different ways, they are all simply dependencies between tasks.

### 43.10.1. Dependencies Internal to the .bb File¶

BitBake uses the `addtask` directive to manage dependencies that are internal to a given recipe file. You can use the `addtask` directive to indicate when a task is dependent on other tasks or when other tasks depend on that recipe. Here is an example:

```
addtask printdate after do_fetch before do_build
```

In this example, the `do_printdate` task depends on the completion of the `do_fetch` task, and the `do_build` task depends on the completion of the `do_printdate` task.

## Note

For a task to run, it must be a direct or indirect dependency of some other task that is scheduled to run.

For illustration, here are some examples:

- The directive `addtask mytask before do_configure` causes `do_mytask` to run before `do_configure` runs. Be aware that `do_mytask` still only runs if its `input checksum` has changed since the last time it was run. Changes to the input checksum of `do_mytask` also indirectly cause `do_configure` to run.
- The directive `addtask mytask after do_configure` by itself never causes `do_mytask` to run. `do_mytask` can still be run manually as follows:

```
$ bitbake recipe -c mytask
```

Declaring `do_mytask` as a dependency of some other task that is scheduled to run also causes it to run. Regardless, the task runs after `do_configure`.

### 43.10.2. Build Dependencies¶

BitBake uses the `DEPENDS` variable to manage build time dependencies. The `[deptask]` varflag for tasks signifies the task of each item listed in `DEPENDS` that must complete before that task can be executed. Here is an example:

```
do_configure[deptask] = "do_populate_sysroot"
```

In this example, the `do_populate_sysroot` task of each item in `DEPENDS` must complete before `do_configure` can execute.

### 43.10.3. Runtime Dependencies¶

BitBake uses the `PACKAGES`, `RDEPENDS`, and `RRECOMMENDS` variables to manage runtime dependencies.

The `PACKAGES` variable lists runtime packages. Each of those packages can have `RDEPENDS` and `RRECOMMENDS` runtime dependencies. The `[rdeptask]` flag for tasks is used to signify the task of each item runtime dependency which must have completed before that task can be executed.

```
do_package_qa[rdeptask] = "do_packagedata"
```

In the previous example, the `do_packagedata` task of each item in `RDEPENDS` must have completed before `do_package_qa` can execute.

### 43.10.4. Recursive Dependencies¶

BitBake uses the `[recrdeptask]` flag to manage recursive task dependencies. BitBake looks through the build-time and runtime dependencies of the current recipe, looks through the task's inter-task dependencies, and then adds dependencies for the listed task. Once BitBake has accomplished this, it recursively works through the dependencies of those tasks. Iterative passes continue until all dependencies are discovered and added.

The `[recrdeptask]` flag is most commonly used in high-level recipes that need to wait for some task to finish "globally". For example, `image.bbclass` has the following:

```
do_rootfs[recrdeptask] += "do_packagedata"
```

This statement says that the `do_packagedata` task of the current recipe and all recipes reachable (by way of dependencies) from the `image` recipe must run before the `do_rootfs` task can run.

You might want to not only have BitBake look for dependencies of those tasks, but also have BitBake look for build-time and runtime dependencies of the dependent tasks as well. If that is the case, you need to reference the task name itself in the task list:

```
do_a[recrdeptask] = "do_a do_b"
```

### 43.10.5. Inter-Task Dependencies¶

BitBake uses the `[depends]` flag in a more generic form to manage inter-task dependencies. This more generic form allows for inter-dependency checks for specific tasks rather than checks for the data in `DEPENDS`. Here is an example:

```
do_patch[depends] = "quilt-native:do_populate_sysroot"
```

In this example, the `do_populate_sysroot` task of the target `quilt-native` must have completed before the `do_patch` task can execute.

The `[rdepends]` flag works in a similar way but takes targets in the runtime namespace instead of the build-time dependency namespace.

## 43.11. Functions You Can Call From Within Python¶

BitBake provides many functions you can call from within Python functions. This section lists the most commonly used functions, and mentions where to find others.

### 43.11.1. Functions for Accessing Datastore Variables¶

It is often necessary to access variables in the BitBake datastore using Python functions. The Bitbake datastore has an API that allows you this access. Here is a list of available operations:

| <i>Operation</i>                                  | <i>Description</i>                                                                                                                                                                                                                                      |
|---------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>d.getVar("X", expand)</code>                | Returns the value of variable "X". Using "expand=True" expands the value. Returns "None" if the variable "X" does not exist.                                                                                                                            |
| <code>d.setVar("X", "value")</code>               | Sets the variable "X" to "value".                                                                                                                                                                                                                       |
| <code>d.appendVar("X", "value")</code>            | Adds "value" to the end of the variable "X". Acts like <code>d.setVar("X", "value")</code> if the variable "X" does not exist.                                                                                                                          |
| <code>d.prependVar("X", "value")</code>           | Adds "value" to the start of the variable "X". Acts like <code>d.setVar("X", "value")</code> if the variable "X" does not exist.                                                                                                                        |
| <code>d.delVar("X")</code>                        | Deletes the variable "X" from the datastore. Does nothing if the variable "X" does not exist.                                                                                                                                                           |
| <code>d.renameVar("X", "Y")</code>                | Renames the variable "X" to "Y". Does nothing if the variable "X" does not exist.                                                                                                                                                                       |
| <code>d.getVarFlag("X", flag, expand)</code>      | Returns the value of variable "X". Using "expand=True" expands the value. Returns "None" if either the variable "X" or the named flag does not exist.                                                                                                   |
| <code>d.setVarFlag("X", flag, "value")</code>     | Sets the named flag for variable "X" to "value".                                                                                                                                                                                                        |
| <code>d.appendVarFlag("X", flag, "value")</code>  | Appends "value" to the named flag on the variable "X". Acts like <code>d.setVarFlag("X", flag, "value")</code> if the named flag does not exist.                                                                                                        |
| <code>d.prependVarFlag("X", flag, "value")</code> | Prepends "value" to the named flag on the variable "X". Acts like <code>d.setVarFlag("X", flag, "value")</code> if the named flag does not exist.                                                                                                       |
| <code>d.delVarFlag("X", flag)</code>              | Deletes the named flag on the variable "X" from the datastore.                                                                                                                                                                                          |
| <code>d.setVarFlags("X", flagsdict)</code>        | Sets the flags specified in the <code>flagsdict()</code> parameter. <code>setVarFlags</code> does not clear previous flags. Think of this operation as <code>addVarFlags</code> .                                                                       |
| <code>d.getVarFlags("X")</code>                   | Returns a <code>flagsdict</code> of the flags for the variable "X". Returns "None" if the variable "X" does not exist.                                                                                                                                  |
| <code>d.delVarFlags("X")</code>                   | Deletes all the flags for the variable "X". Does nothing if the variable "X" does not exist.                                                                                                                                                            |
| <code>d.expand(expression)</code>                 | Expands variable references in the specified string expression. References to variables that do not exist are left as is. For example, <code>d.expand("foo \${X}")</code> expands to the literal string "foo \${X}" if the variable "X" does not exist. |

### 43.11.2. Other Functions¶

You can find many other functions that can be called from Python by looking at the source code of the `bb` module, which is in `bitbake/lib/bb`. For example, `bitbake/lib/bb/utils.py` includes the commonly used functions `bb.utils.contains()` and `bb.utils.mkdirhier()`, which come with docstrings.

## 43.12. Task Checksums and Setscene¶

BitBake uses checksums (or signatures) along with the setscene to determine if a task needs to be run. This section describes the process. To help understand how BitBake does this, the section assumes an OpenEmbedded metadata-based example.

These checksums are stored in [STAMP](#). You can examine the checksums using the following BitBake command:

```
$ bitbake-dumpsigs
```

This command returns the signature data in a readable format that allows you to examine the inputs used when the OpenEmbedded build system generates signatures. For example, using `bitbake-dumpsigs` allows you to examine the `do_compile` task's "sigdata" for a C application (e.g. `bash`). Running the command also reveals that the "CC" variable is part of the inputs that are hashed. Any changes to this variable would invalidate the stamp and cause the `do_compile` task to run.

The following list describes related variables:

- [BB\\_HASHCHECK\\_FUNCTION](#): Specifies the name of the function to call during the "setscene" part of the task's execution in order to validate the list of task hashes.
- [BB\\_SETSCENE\\_DEPVALID](#): Specifies a function BitBake calls that determines whether BitBake requires a setscene dependency to be met.
- [BB\\_SETSCENE\\_VERIFY\\_FUNCTION2](#): Specifies a function to call that verifies the list of planned task execution before the main task execution happens.
- [BB\\_STAMP\\_POLICY](#): Defines the mode for comparing timestamps of stamp files.
- [BB\\_STAMP\\_WHITELIST](#): Lists stamp files that are looked at when the stamp policy is "whitelist".
- [BB\\_TASKHASH](#): Within an executing task, this variable holds the hash of the task as returned by the currently enabled signature generator.
- [STAMP](#): The base path to create stamp files.
- [STAMPCLEAN](#): Again, the base path to create stamp files but can use wildcards for matching a range of files for clean operations.

## 43.13. Wildcard Support in Variables¶

Support for wildcard use in variables varies depending on the context in which it is used. For example, some variables and file names allow limited use of wildcards through the "%" and "\*" characters. Other variables or names support Python's `glob` syntax, `fnmatch` syntax, or `Regular Expression (re)` syntax.

For variables that have wildcard support, the documentation describes which form of wildcard, its use, and its limitations.

## Chapter 44. File Download Support

[44.1. The Download \(Fetch\).](#)

[44.2. The Unpack](#)

[44.3. Fetchers](#)

[44.3.1. Local file fetcher \(file://\).](#)

[44.3.2. HTTP/FTP wget fetcher \(http://, ftp://, https://\).](#)

[44.3.3. CVS fetcher \(cvs://\).](#)

[44.3.4. Subversion \(SVN\) Fetcher \(svn://\).](#)

[44.3.5. Git Fetcher \(git://\).](#)

[44.3.6. Git Submodule Fetcher \(git-sm://\).](#)

[44.3.7. ClearCase Fetcher \(ccrc://\).](#)

[44.3.8. Perforce Fetcher \(p4://\).](#)

[44.3.9. Repo Fetcher \(repo://\).](#)

[44.3.10. Other Fetchers](#)

[44.4. Auto Revisions](#)

BitBake's fetch module is a standalone piece of library code that deals with the intricacies of downloading source code and files from remote systems. Fetching source code is one of the cornerstones of building software. As such, this module forms an important part of BitBake.

The current fetch module is called "fetch2" and refers to the fact that it is the second major version of the API. The original version is obsolete and has been removed from the codebase. Thus, in all cases, "fetch" refers to "fetch2" in this manual.

### 44.1. The Download (Fetch)

BitBake takes several steps when fetching source code or files. The fetcher codebase deals with two distinct processes in order: obtaining the files from somewhere (cached or otherwise) and then unpacking those files into a specific location and perhaps in a specific way. Getting and unpacking the files is often optionally followed by patching. Patching, however, is not covered by this module.

The code to execute the first part of this process, a fetch, looks something like the following:

```
src_uri = (d.getVar('SRC_URI') or "").split()
fetcher = bb.fetch2.Fetch(src_uri, d)
fetcher.download()
```

This code sets up an instance of the fetch class. The instance uses a space-separated list of URLs from the `SRC_URI` variable and then calls the `download` method to download the files.

The instantiation of the fetch class is usually followed by:

```
rootdir = l.getVar('WORKDIR')
fetcher.unpack(rootdir)
```

This code unpacks the downloaded files to the specified by `WORKDIR`.

#### Note

For convenience, the naming in these examples matches the variables used by OpenEmbedded. If you want to see the above code in action, examine the OpenEmbedded class file `base.bbclass`.

The `SRC_URI` and `WORKDIR` variables are not hardcoded into the fetcher, since those fetcher methods can be (and are) called with different variable names. In OpenEmbedded for example, the shared state (sstate) code uses the fetch module to fetch the sstate files.

When the `download()` method is called, BitBake tries to resolve the URLs by looking for source files in a specific search order:

- **Pre-mirror Sites:** BitBake first uses pre-mirrors to try and find source files. These locations are defined using the `PREMIRRORS` variable.
- **Source URI:** If pre-mirrors fail, BitBake uses the original URL (e.g from `SRC_URI`).
- **Mirror Sites:** If fetch failures occur, BitBake next uses mirror locations as defined by the `MIRRORS` variable.

For each URL passed to the fetcher, the fetcher calls the submodule that handles that particular URL type. This behavior can be the source of some confusion when you are providing URLs for the `SRC_URI` variable. Consider the following two URLs:

```
http://git.yoctoproject.org/git/poky;protocol=git
git://git.yoctoproject.org/git/poky;protocol=http
```

In the former case, the URL is passed to the `wget` fetcher, which does not understand "git". Therefore, the latter case is the correct form since the Git fetcher does know how to use HTTP as a transport.

Here are some examples that show commonly used mirror definitions:

```

PREMIRRORS ?= "\
bzip://.*.* http://somemirror.org/sources/ \n \
cvs://.*.* http://somemirror.org/sources/ \n \
git://.*.* http://somemirror.org/sources/ \n \
hg://.*.* http://somemirror.org/sources/ \n \
osc://.*.* http://somemirror.org/sources/ \n \
p4://.*.* http://somemirror.org/sources/ \n \
svn://.*.* http://somemirror.org/sources/ \n"

MIRRORS += "\
ftp://.*.* http://somemirror.org/sources/ \n \
http://.*.* http://somemirror.org/sources/ \n \
https://.*.* http://somemirror.org/sources/ \n"

```

It is useful to note that BitBake supports cross-URLs. It is possible to mirror a Git repository on an HTTP server as a tarball. This is what the `git://` mapping in the previous example does.

Since network accesses are slow, BitBake maintains a cache of files downloaded from the network. Any source files that are not local (i.e. downloaded from the Internet) are placed into the download directory, which is specified by the `DL_DIR` variable.

File integrity is of key importance for reproducing builds. For non-local archive downloads, the fetcher code can verify SHA-256 and MD5 checksums to ensure the archives have been downloaded correctly. You can specify these checksums by using the `SRC_URI` variable with the appropriate varflags as follows:

```

SRC_URI[md5sum] = "value"
SRC_URI[sha256sum] = "value"

```

You can also specify the checksums as parameters on the `SRC_URI` as shown below:

```

SRC_URI = "http://example.com/foobar.tar.bz2;md5sum=4a8e0f237e961fd7785d19d07fdb994d"

```

If multiple URIs exist, you can specify the checksums either directly as in the previous example, or you can name the URLs. The following syntax shows how you name the URIs:

```

SRC_URI = "http://example.com/foobar.tar.bz2;name=foo"
SRC_URI[foo.md5sum] = 4a8e0f237e961fd7785d19d07fdb994d

```

After a file has been downloaded and has had its checksum checked, a ".done" stamp is placed in `DL_DIR`. BitBake uses this stamp during subsequent builds to avoid downloading or comparing a checksum for the file again.

## Note

It is assumed that local storage is safe from data corruption. If this were not the case, there would be bigger issues to worry about.

If `BB_STRICT_CHECKSUM` is set, any download without a checksum triggers an error message. The `BB_NO_NETWORK` variable can be used to make any attempted network access a fatal error, which is useful for checking that mirrors are complete as well as other things.

## 44.2. The Unpack

The unpack process usually immediately follows the download. For all URLs except Git URLs, BitBake uses the common `unpack` method.

A number of parameters exist that you can specify within the URL to govern the behavior of the unpack stage:

- **unpack:** Controls whether the URL components are unpacked. If set to "1", which is the default, the components are unpacked. If set to "0", the unpack stage leaves the file alone. This parameter is useful when you want an archive to be copied in and not be unpacked.
- **dos:** Applies to `.zip` and `.jar` files and specifies whether to use DOS line ending conversion on text files.
- **basepath:** Instructs the unpack stage to strip the specified directories from the source path when unpacking.
- **subdir:** Unpacks the specific URL to the specified subdirectory within the root directory.

The unpack call automatically decompresses and extracts files with ".Z", ".z", ".gz", ".xz", ".zip", ".jar", ".ipk", ".rpm", ".srpm", ".deb" and ".bz2" extensions as well as various combinations of tarball extensions.

As mentioned, the Git fetcher has its own unpack method that is optimized to work with Git trees. Basically, this method works by cloning the tree into the final directory. The process is completed using references so that there is only one central copy of the Git metadata needed.

## 44.3. Fetchers

As mentioned earlier, the URL prefix determines which fetcher submodule BitBake uses. Each submodule can support different URL parameters, which are described in the following sections.

### 44.3.1. Local file fetcher (`file://`)

This submodule handles URLs that begin with `file://`. The filename you specify within the URL can be either an absolute or relative path to a file. If the filename is relative, the contents of the `FILES_PATH` variable is used in the same way `PATH` is used to find executables. If the file cannot be found, it is assumed that it is available in `DL_DIR` by the time the `download()` method is called.

If you specify a directory, the entire directory is unpacked.

Here are a couple of example URLs, the first relative and the second absolute:

```
SRC_URI = "file://relativefile.patch"
SRC_URI = "file:///Users/ich/very_important_software"
```

#### 44.3.2. HTTP/FTP wget fetcher (`http://`, `ftp://`, `https://`)¶

This fetcher obtains files from web and FTP servers. Internally, the fetcher uses the `wget` utility.

The executable and parameters used are specified by the `FETCHCMD_wget` variable, which defaults to sensible values. The fetcher supports a parameter "downloadfilename" that allows the name of the downloaded file to be specified. Specifying the name of the downloaded file is useful for avoiding collisions in `DL_DIR` when dealing with multiple files that have the same name.

Some example URLs are as follows:

```
SRC_URI = "http://oe.handhelds.org/not_there.aac"
SRC_URI = "ftp://oe.handhelds.org/not_there_as_well.aac"
SRC_URI = "ftp://you@oe.handhelds.org/home/you/secret.plan"
```

#### Note

Because URL parameters are delimited by semi-colons, this can introduce ambiguity when parsing URLs that also contain semi-colons, for example:

```
SRC_URI = "http://abc123.org/git/?p=gcc/gcc.git;a=snapshot;h=a5dd47"
```

Such URLs should be modified by replacing semi-colons with '&' characters:

```
SRC_URI = "http://abc123.org/git/?p=gcc/gcc.git&a=snapshot&h=a5dd47"
```

In most cases this should work. Treating semi-colons and '&' in queries identically is recommended by the World Wide Web Consortium (W3C). Note that due to the nature of the URL, you may have to specify the name of the downloaded file as well:

```
SRC_URI = "http://abc123.org/git/?p=gcc/gcc.git&a=snapshot&h=a5dd47;downloadfilename=myfile.bz2"
```

#### 44.3.3. CVS fetcher (`cvs://`)¶

This submodule handles checking out files from the CVS version control system. You can configure it using a number of different variables:

- **`FETCHCMD_cvs`**: The name of the executable to use when running the `CVS` command. This name is usually "cvs".
- **`SRCDATE`**: The date to use when fetching the CVS source code. A special value of "now" causes the checkout to be updated on every build.
- **`CVSDIR`**: Specifies where a temporary checkout is saved. The location is often `DL_DIR/cvs`.
- **`CVS_PROXY_HOST`**: The name to use as a "proxy=" parameter to the `CVS` command.
- **`CVS_PROXY_PORT`**: The port number to use as a "proxyport=" parameter to the `CVS` command.

As well as the standard username and password URL syntax, you can also configure the fetcher with various URL parameters:

The supported parameters are as follows:

- **`"method"`**: The protocol over which to communicate with the CVS server. By default, this protocol is "pserver". If "method" is set to "ext", BitBake examines the "rsh" parameter and sets `CVS_RSH`. You can use "dir" for local directories.
- **`"module"`**: Specifies the module to check out. You must supply this parameter.
- **`"tag"`**: Describes which CVS TAG should be used for the checkout. By default, the TAG is empty.
- **`"date"`**: Specifies a date. If no "date" is specified, the `SRCDATE` of the configuration is used to checkout a specific date. The special value of "now" causes the checkout to be updated on every build.
- **`"localdir"`**: Used to rename the module. Effectively, you are renaming the output directory to which the module is unpacked. You are forcing the module into a special directory relative to `CVSDIR`.
- **`"rsh"`**: Used in conjunction with the "method" parameter.
- **`"scmdata"`**: Causes the CVS metadata to be maintained in the tarball the fetcher creates when set to "keep". The tarball is expanded into the work directory. By default, the CVS metadata is removed.
- **`"fullpath"`**: Controls whether the resulting checkout is at the module level, which is the default, or is at deeper paths.
- **`"norecurse"`**: Causes the fetcher to only checkout the specified directory with no recurse into any subdirectories.
- **`"port"`**: The port to which the CVS server connects.

Some example URLs are as follows:

```
SRC_URI = "cvs://CVSROOT;module=mymodule;tag=some-version;method=ext"
SRC_URI = "cvs://CVSROOT;module=mymodule;date=20060126;localdir=usethat"
```



#### 44.3.4. Subversion (SVN) Fetcher (`svn://`)

This fetcher submodule fetches code from the Subversion source control system. The executable used is specified by `FETCHCMD_svn`, which defaults to "svn". The fetcher's temporary working directory is set by `SVN_DIR`, which is usually `DL_DIR/svn`.

The supported parameters are as follows:

- **"module"**: The name of the svn module to checkout. You must provide this parameter. You can think of this parameter as the top-level directory of the repository data you want.
- **"path\_spec"**: A specific directory in which to checkout the specified svn module.
- **"protocol"**: The protocol to use, which defaults to "svn". If "protocol" is set to "svn+ssh", the "ssh" parameter is also used.
- **"rev"**: The revision of the source code to checkout.
- **"scmdata"**: Causes the ".svn" directories to be available during compile-time when set to "keep". By default, these directories are removed.
- **"ssh"**: An optional parameter used when "protocol" is set to "svn+ssh". You can use this parameter to specify the ssh program used by svn.
- **"transportuser"**: When required, sets the username for the transport. By default, this parameter is empty. The transport username is different than the username used in the main URL, which is passed to the subversion command.

Following are three examples using svn:

```
SRC_URI = "svn://myrepos/proj1;module=vip;protocol=http;rev=667"
SRC_URI = "svn://myrepos/proj1;module=opie;protocol=svn+ssh"
SRC_URI = "svn://myrepos/proj1;module=trunk;protocol=http;path_spec=${MY_DIR}/proj1"
```

#### 44.3.5. Git Fetcher (`git://`)

This fetcher submodule fetches code from the Git source control system. The fetcher works by creating a bare clone of the remote into `GIT_DIR`, which is usually `DL_DIR/git2`. This bare clone is then cloned into the work directory during the unpack stage when a specific tree is checked out. This is done using alternates and by reference to minimize the amount of duplicate data on the disk and make the unpack process fast. The executable used can be set with `FETCHCMD_git`.

This fetcher supports the following parameters:

- **"protocol"**: The protocol used to fetch the files. The default is "git" when a hostname is set. If a hostname is not set, the Git protocol is "file". You can also use "http", "https", "ssh" and "rsync".
- **"nocheckout"**: Tells the fetcher to not checkout source code when unpacking when set to "1". Set this option for the URL where there is a custom routine to checkout code. The default is "0".
- **"rebaseable"**: Indicates that the upstream Git repository can be rebased. You should set this parameter to "1" if revisions can become detached from branches. In this case, the source mirror tarball is done per revision, which has a loss of efficiency. Rebasing the upstream Git repository could cause the current revision to disappear from the upstream repository. This option reminds the fetcher to preserve the local cache carefully for future use. The default value for this parameter is "0".
- **"nobranch"**: Tells the fetcher to not check the SHA validation for the branch when set to "1". The default is "0". Set this option for the recipe that refers to the commit that is valid for a tag instead of the branch.
- **"bareclone"**: Tells the fetcher to clone a bare clone into the destination directory without checking out a working tree. Only the raw Git metadata is provided. This parameter implies the "nocheckout" parameter as well.
- **"branch"**: The branch(es) of the Git tree to clone. If unset, this is assumed to be "master". The number of branch parameters much match the number of name parameters.
- **"rev"**: The revision to use for the checkout. The default is "master".
- **"tag"**: Specifies a tag to use for the checkout. To correctly resolve tags, BitBake must access the network. For that reason, tags are often not used. As far as Git is concerned, the "tag" parameter behaves effectively the same as the "rev" parameter.
- **"subpath"**: Limits the checkout to a specific subpath of the tree. By default, the whole tree is checked out.
- **"destsuffix"**: The name of the path in which to place the checkout. By default, the path is `git/`.
- **"usehead"**: Enables local `git://` URLs to use the current branch HEAD as the revision for use with `AUTOREV`. The "usehead" parameter implies no branch and only works when the transfer protocol is `file://`.

Here are some example URLs:

```
SRC_URI = "git://git.oe.handhelds.org/git/vip.git;tag=version-1"
SRC_URI = "git://git.oe.handhelds.org/git/vip.git;protocol=http"
```

#### 44.3.6. Git Submodule Fetcher (`git:sm://`)

This fetcher submodule inherits from the [Git fetcher](#) and extends that fetcher's behavior by fetching a repository's submodules. `SRC_URI` is passed to the Git fetcher as described in the "[Git Fetcher \(git://\)](#)" section.

### Notes and Warnings

You must clean a recipe when switching between `'git://'` and `'git:sm://'` URLs.

The Git Submodules fetcher is not a complete fetcher implementation. The fetcher has known issues where it does not use the normal source mirroring infrastructure properly. Further, the submodule sources it fetches



are not visible to the licensing and source archiving infrastructures.

#### 44.3.7. ClearCase Fetcher (`ccrc : //`)

This fetcher submodule fetches code from a [ClearCase](#) repository.

To use this fetcher, make sure your recipe has proper `SRC_URI`, `SRCREV`, and `PV` settings. Here is an example:

```
SRC_URI = "ccrc://cc.example.org/ccrc;vob=/example_vob;module=/example_module"
SRCREV = "EXAMPLE_CLEARCASE_TAG"
PV = "${@d.getVar("SRCREV", False).replace("/", "+")}"
```

The fetcher uses the `rcleartool` or `cleartool` remote client, depending on which one is available.

Following are options for the `SRC_URI` statement:

- **vob**: The name, which must include the prepending `/` character, of the ClearCase VOB. This option is required.
- **module**: The module, which must include the prepending `/` character, in the selected VOB.

#### Note

The `module` and `vob` options are combined to create the `load` rule in the view config spec. As an example, consider the `vob` and `module` values from the `SRC_URI` statement at the start of this section. Combining those values results in the following:

```
load /example_vob/example_module
```

- **proto**: The protocol, which can be either `http` or `https`.

By default, the fetcher creates a configuration specification. If you want this specification written to an area other than the default, use the `CCASE_CUSTOM_CONFIG_SPEC` variable in your recipe to define where the specification is written.

#### Note

the `SRCREV` loses its functionality if you specify this variable. However, `SRCREV` is still used to label the archive after a fetch even though it does not define what is fetched.

Here are a couple of other behaviors worth mentioning:

- When using `cleartool`, the login of `cleartool` is handled by the system. The login require no special steps.
- In order to use `rcleartool` with authenticated users, an "rcleartool login" is necessary before using the fetcher.

#### 44.3.8. Perforce Fetcher (`p4 : //`)

This fetcher submodule fetches code from the [Perforce](#) source control system. The executable used is specified by `FETCHCMD_p4`, which defaults to `"p4"`. The fetcher's temporary working directory is set by `P4DIR`, which defaults to `"DL_DIR/p4"`.

To use this fetcher, make sure your recipe has proper `SRC_URI`, `SRCREV`, and `PV` values. The `p4` executable is able to use the config file defined by your system's `P4CONFIG` environment variable in order to define the Perforce server URL and port, username, and password if you do not wish to keep those values in a recipe itself. If you choose not to use `P4CONFIG`, or to explicitly set variables that `P4CONFIG` can contain, you can specify the `P4PORT` value, which is the server's URL and port number, and you can specify a username and password directly in your recipe within `SRC_URI`.

Here is an example that relies on `P4CONFIG` to specify the server URL and port, username, and password, and fetches the Head Revision:

```
SRC_URI = "p4://example-depot/main/source/..."
SRCREV = "${AUTOREV}"
PV = "p4-${SRCPV}"
S = "${WORKDIR}/p4"
```

Here is an example that specifies the server URL and port, username, and password, and fetches a Revision based on a Label:

```
P4PORT = "tcp:p4server.example.net:1666"
SRC_URI = "p4://user:passwd@example-depot/main/source/..."
SRCREV = "release-1.0"
PV = "p4-${SRCPV}"
S = "${WORKDIR}/p4"
```

#### Note

You should always set `S` to `"${WORKDIR}/p4"` in your recipe.

#### 44.3.9. Repo Fetcher (`repo : //`)

This fetcher submodule fetches code from `google-repo` source control system. The fetcher works by initiating and syncing sources of the repository into `REPODIR`, which is usually `DL_DIR/repo`.

This fetcher supports the following parameters:

- **"protocol"**: Protocol to fetch the repository manifest (default: `git`).
- **"branch"**: Branch or tag of repository to get (default: `master`).
- **"manifest"**: Name of the manifest file (default: `default.xml`).

Here are some example URLs:

```
SRC_URI = "repo://REPOROOT;protocol=git;branch=some_branch;manifest=my_manifest.xml"
SRC_URI = "repo://REPOROOT;protocol=file;branch=some_branch;manifest=my_manifest.xml"
```

44.3.10. Other Fetchers¶

Fetch submodules also exist for the following:

- Bazaar (`bzr://`)
- Mercurial (`hg://`)
- npm (`npm://`)
- OSC (`osc://`)
- Secure FTP (`sftp://`)
- Secure Shell (`ssh://`)
- Trees using Git Annex (`gitannex://`)

No documentation currently exists for these lesser used fetcher submodules. However, you might find the code helpful and readable.

44.4. Auto Revisions¶

We need to document `AUTOREV` and `SRCREV_FORMAT` here.

Chapter 45. Variables Glossary¶

Glossary

This chapter lists common variables used by BitBake and gives an overview of their function and contents.

Note

Following are some points regarding the variables listed in this glossary:

- The variables listed in this glossary are specific to BitBake. Consequently, the descriptions are limited to that context.
- Also, variables exist in other systems that use BitBake (e.g. The Yocto Project and OpenEmbedded) that have names identical to those found in this glossary. For such cases, the variables in those systems extend the functionality of the variable as it is described here in this glossary.
- Finally, there are variables mentioned in this glossary that do not appear in the BitBake glossary. These other variables are variables used in systems that use BitBake.

Glossary¶

A B C D E F G H I L M O P R S T

A

ASSUME\_PROVIDED¶

Lists recipe names (`PN` values) BitBake does not attempt to build. Instead, BitBake assumes these recipes have already been built.

In OpenEmbedded-Core, `ASSUME_PROVIDED` mostly specifies native tools that should not be built. An example is `git-native`, which when specified allows for the Git binary from the host to be used rather than building `git-native`.

B

B¶

The directory in which BitBake executes functions during a recipe's build process.

BB\_ALLOWED\_NETWORKS¶

Specifies a space-delimited list of hosts that the fetcher is allowed to use to obtain the required source code. Following are considerations surrounding this variable:

- This host list is only used if `BB_NO_NETWORK` is either not set or set to "0".
- Limited support for the "\*" wildcard character for matching against the beginning of host names exists. For example, the following setting matches `git.gnu.org`, `ftp.gnu.org`, and `foo.git.gnu.org`.

```
BB_ALLOWED_NETWORKS = "*.gnu.org"
```

## Important

The use of the "\*" character only works at the beginning of a host name and it must be isolated from the remainder of the host name. You cannot use the wildcard character in any other location of the name or combined with the front part of the name.

For example, `*.foo.bar` is supported, while `*aa.foo.bar` is not.

- Mirrors not in the host list are skipped and logged in debug.
- Attempts to access networks not in the host list cause a failure.

Using `BB_ALLOWED_NETWORKS` in conjunction with `PREMIRRORS` is very useful. Adding the host you want to use to `PREMIRRORS` results in the source code being fetched from an allowed location and avoids raising an error when a host that is not allowed is in a `SRC_URI` statement. This is because the fetcher does not attempt to use the host listed in `SRC_URI` after a successful fetch from the `PREMIRRORS` occurs.

`BB_CONSOLELOG`

Specifies the path to a log file into which BitBake's user interface writes output during the build.

`BB_CURRENTTASK`

Contains the name of the currently running task. The name does not include the `do_` prefix.

`BB_DANGLINGAPPENDS_WARNONLY`

Defines how BitBake handles situations where an append file (`.bbappend`) has no corresponding recipe file (`.bb`). This condition often occurs when layers get out of sync (e.g. `oe-core` bumps a recipe version and the old recipe no longer exists and the other layer has not been updated to the new version of the recipe yet).

The default fatal behavior is safest because it is the same reaction given something is out of sync. It is important to realize when your changes are no longer being applied.

`BB_DEFAULT_TASK`

The default task to use when none is specified (e.g. with the `-C` command line option). The task name specified should not include the `do_` prefix.

`BB_DISKMON_DIRS`

Monitors disk space and available inodes during the build and allows you to control the build based on these parameters.

Disk space monitoring is disabled by default. When setting this variable, use the following form:

```
BB_DISKMON_DIRS = "<action>,<dir>,<threshold> [...]"
```

where:

`<action>` is:

**ABORT:** Immediately abort the build when a threshold is broken.

**STOPTASKS:** Stop the build after the currently executing tasks have finished when a threshold is broken.

**WARN:** Issue a warning but continue the build when a threshold is broken. Subsequent warnings are issued as defined by the `BB_DISKMON_WARNINTERVAL` variable, which must be defined.

`<dir>` is:

Any directory you choose. You can specify one or more directories to monitor by separating the groupings with a space. If two directories are on the same device, only the first directory is monitored.

`<threshold>` is:

Either the minimum available disk space, the minimum number of free inodes, or both. You must specify at least one. To omit one or the other, simply omit the value. Specify the threshold using G, M, K for Gbytes, Mbytes, and Kbytes, respectively. If you do not specify G, M, or K, Kbytes is assumed by default. Do not use GB, MB, or KB.

Here are some examples:

```
BB_DISKMON_DIRS = "ABORT,${TMPDIR},1G,100K WARN,${SSTATE_DIR},1G,100K"
BB_DISKMON_DIRS = "STOPTASKS,${TMPDIR},1G,100K"
```

```
BB_DISKMON_DIRS = "${TMPDIR},1G,100K"
BB_DISKMON_DIRS = "ABORT,${TMPDIR},,100K"
```

The first example works only if you also set the `BB_DISKMON_WARNINTERVAL` variable. This example causes the build system to immediately abort when either the disk space in `${TMPDIR}` drops below 1 Gbyte or the available free inodes drops below 100 Kbytes. Because two directories are provided with the variable, the build system also issues a warning when the disk space in the `${SSTATE_DIR}` directory drops below 1 Gbyte or the number of free inodes drops below 100 Kbytes. Subsequent warnings are issued during intervals as defined by the `BB_DISKMON_WARNINTERVAL` variable.

The second example stops the build after all currently executing tasks complete when the minimum disk space in the `${TMPDIR}` directory drops below 1 Gbyte. No disk monitoring occurs for the free inodes in this case.

The final example immediately aborts the build when the number of free inodes in the `${TMPDIR}` directory drops below 100 Kbytes. No disk space monitoring for the directory itself occurs in this case.

`BB_DISKMON_WARNINTERVAL`

Defines the disk space and free inode warning intervals.

If you are going to use the `BB_DISKMON_WARNINTERVAL` variable, you must also use the `BB_DISKMON_DIRS` variable and define its action as "WARN". During the build, subsequent warnings are issued each time disk space or number of free inodes further reduces by the respective interval.

If you do not provide a `BB_DISKMON_WARNINTERVAL` variable and you do use `BB_DISKMON_DIRS` with the "WARN" action, the disk monitoring interval defaults to the following:

```
BB_DISKMON_WARNINTERVAL = "50M,5K"
```

When specifying the variable in your configuration file, use the following form:

```
BB_DISKMON_WARNINTERVAL = "<disk_space_interval>,<disk_inode_interval>"
```

where:

`<disk_space_interval>` is:  
An interval of memory expressed in either G, M, or K for Gbytes, Mbytes, or Kbytes, respectively. You cannot use GB, MB, or KB.

`<disk_inode_interval>` is:  
An interval of free inodes expressed in either G, M, or K for Gbytes, Mbytes, or Kbytes, respectively. You cannot use GB, MB, or KB.

Here is an example:

```
BB_DISKMON_DIRS = "WARN,${SSTATE_DIR},1G,100K"
BB_DISKMON_WARNINTERVAL = "50M,5K"
```

These variables cause BitBake to issue subsequent warnings each time the available disk space further reduces by 50 Mbytes or the number of free inodes further reduces by 5 Kbytes in the `${SSTATE_DIR}` directory. Subsequent warnings based on the interval occur each time a respective interval is reached beyond the initial warning (i.e. 1 Gbytes and 100 Kbytes).

`BB_ENV_WHITELIST`

Specifies the internal whitelist of variables to allow through from the external environment into BitBake's datastore. If the value of this variable is not specified (which is the default), the following list is used: `BBPATH`, `BB_PRESERVE_ENV`, `BB_ENV_WHITELIST`, and `BB_ENV_EXTRAWHITE`.

## Note

You must set this variable in the external environment in order for it to work.

`BB_ENV_EXTRAWHITE`

Specifies an additional set of variables to allow through (whitelist) from the external environment into BitBake's datastore. This list of variables are on top of the internal list set in `BB_ENV_WHITELIST`.

## Note

You must set this variable in the external environment in order for it to work.

`BB_FETCH_PREMIRRORONLY`

When set to "1", causes BitBake's fetcher module to only search `PREMIRRORS` for files. BitBake will not search the main `SRC_URI` or `MIRRORS`.

`BB_FILENAME`

**BB\_GENERATE\_MIRROR\_TARBALLS¶**

Contains the filename of the recipe that owns the currently running task. For example, if the `do_fetch` task that resides in the `my-recipe.bb` is executing, the `BB_FILENAME` variable contains `"/foo/path/my-recipe.bb"`.

Causes tarballs of the Git repositories, including the Git metadata, to be placed in the `DL_DIR` directory. Anyone wishing to create a source mirror would want to enable this variable.

For performance reasons, creating and placing tarballs of the Git repositories is not the default action by BitBake.

```
BB_GENERATE_MIRROR_TARBALLS = "1"
```

**BB\_HASHCONFIG\_WHITELIST¶**

Lists variables that are excluded from base configuration checksum, which is used to determine if the cache can be reused.

One of the ways BitBake determines whether to re-parse the main metadata is through checksums of the variables in the datastore of the base configuration data. There are variables that you typically want to exclude when checking whether or not to re-parse and thus rebuild the cache. As an example, you would usually exclude `TIME` and `DATE` because these variables are always changing. If you did not exclude them, BitBake would never reuse the cache.

**BB\_HASHBASE\_WHITELIST¶**

Lists variables that are excluded from checksum and dependency data. Variables that are excluded can therefore change without affecting the checksum mechanism. A common example would be the variable for the path of the build. BitBake's output should not (and usually does not) depend on the directory in which it was built.

**BB\_HASHCHECK\_FUNCTION¶**

Specifies the name of the function to call during the "setscene" part of the task's execution in order to validate the list of task hashes. The function returns the list of setscene tasks that should be executed.

At this point in the execution of the code, the objective is to quickly verify if a given setscene function is likely to work or not. It's easier to check the list of setscene functions in one pass than to call many individual tasks. The returned list need not be completely accurate. A given setscene task can still later fail. However, the more accurate the data returned, the more efficient the build will be.

**BB\_INVALIDCONF¶**

Used in combination with the `ConfigParsed` event to trigger re-parsing the base metadata (i.e. all the recipes). The `ConfigParsed` event can set the variable to trigger the re-parse. You must be careful to avoid recursive loops with this functionality.

**BB\_LOGFMT¶**

Specifies the name of the log files saved into `$(T)`. By default, the `BB_LOGFMT` variable is undefined and the log file names get created using the following form:

```
log.{task}.{pid}
```

If you want to force log files to take a specific name, you can set this variable in a configuration file.

**BB\_NICE\_LEVEL¶**

Allows BitBake to run at a specific priority (i.e. nice level). System permissions usually mean that BitBake can reduce its priority but not raise it again. See `BB_TASK_NICE_LEVEL` for additional information.

**BB\_NO\_NETWORK¶**

Disables network access in the BitBake fetcher modules. With this access disabled, any command that attempts to access the network becomes an error.

Disabling network access is useful for testing source mirrors, running builds when not connected to the Internet, and when operating in certain kinds of firewall environments.

**BB\_NUMBER\_THREADS¶**

The maximum number of tasks BitBake should run in parallel at any one time. If your host development system supports multiple cores, a good rule of thumb is to set this variable to twice the number of cores.

**BB\_NUMBER\_PARSE\_THREADS¶**

Sets the number of threads BitBake uses when parsing. By default, the number of threads is equal to the number of cores on the system.

**BB\_ORIGENV¶**

Contains a copy of the original external environment in which BitBake was run. The copy is taken before any whitelisted variable values are filtered into BitBake's datastore.

**Note**

The contents of this variable is a datastore object that can be queried using the normal datastore operations.

**BB\_PRESERVE\_ENV¶**

Disables whitelisting and instead allows all variables through from the external environment into BitBake's datastore.

**Note**

You must set this variable in the external environment in order for it to work.

|                                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>BB_RUNFMT</code>                    | <p>Specifies the name of the executable script files (i.e. run files) saved into <code>\${T}</code>. By default, the <code>BB_RUNFMT</code> variable is undefined and the run file names get created using the following form:</p> <pre>run.{task}.{pid}</pre> <p>If you want to force run files to take a specific name, you can set this variable in a configuration file.</p>                                                                                                                                                                                                                     |
| <code>BB_RUNTASK</code>                   | <p>Contains the name of the currently executing task. The value includes the "do_" prefix. For example, if the currently executing task is <code>do_config</code>, the value is "do_config".</p>                                                                                                                                                                                                                                                                                                                                                                                                     |
| <code>BB_SCHEDULER</code>                 | <p>Selects the name of the scheduler to use for the scheduling of BitBake tasks. Three options exist:</p> <ul style="list-style-type: none"> <li>• <b>basic</b> - The basic framework from which everything derives. Using this option causes tasks to be ordered numerically as they are parsed.</li> <li>• <b>speed</b> - Executes tasks first that have more tasks depending on them. The "speed" option is the default.</li> <li>• <b>completion</b> - Causes the scheduler to try to complete a given recipe once its build has started.</li> </ul>                                             |
| <code>BB_SCHEDULERS</code>                | <p>Defines custom schedulers to import. Custom schedulers need to be derived from the <code>RunQueueScheduler</code> class.</p> <p>For information how to select a scheduler, see the <code>BB_SCHEDULER</code> variable.</p>                                                                                                                                                                                                                                                                                                                                                                        |
| <code>BB_SETSCENE_DEPVALID</code>         | <p>Specifies a function BitBake calls that determines whether BitBake requires a setscene dependency to be met.</p> <p>When running a setscene task, BitBake needs to know which dependencies of that setscene task also need to be run. Whether dependencies also need to be run is highly dependent on the metadata. The function specified by this variable returns a "True" or "False" depending on whether the dependency needs to be met.</p>                                                                                                                                                  |
| <code>BB_SETSCENE_VERIFY_FUNCTION2</code> | <p>Specifies a function to call that verifies the list of planned task execution before the main task execution happens. The function is called once BitBake has a list of setscene tasks that have run and either succeeded or failed.</p> <p>The function allows for a task list check to see if they make sense. Even if BitBake was planning to skip a task, the returned value of the function can force BitBake to run the task, which is necessary under certain metadata defined circumstances.</p>                                                                                          |
| <code>BB_SIGNATURE_EXCLUDE_FLAGS</code>   | <p>Lists variable flags (varflags) that can be safely excluded from checksum and dependency data for keys in the datastore. When generating checksum or dependency data for keys in the datastore, the flags set against that key are normally included in the checksum.</p> <p>For more information on varflags, see the "<a href="#">Variable Flags</a>" section.</p>                                                                                                                                                                                                                              |
| <code>BB_SIGNATURE_HANDLER</code>         | <p>Defines the name of the signature handler BitBake uses. The signature handler defines the way stamp files are created and handled, if and how the signature is incorporated into the stamps, and how the signature itself is generated.</p> <p>A new signature handler can be added by injecting a class derived from the <code>SignatureGenerator</code> class into the global namespace.</p>                                                                                                                                                                                                    |
| <code>BB_SRCREV_POLICY</code>             | <p>Defines the behavior of the fetcher when it interacts with source control systems and dynamic source revisions. The <code>BB_SRCREV_POLICY</code> variable is useful when working without a network.</p> <p>The variable can be set using one of two policies:</p> <ul style="list-style-type: none"> <li>• <b>cache</b> - Retains the value the system obtained previously rather than querying the source control system each time.</li> <li>• <b>clear</b> - Queries the source controls system every time. With this policy, there is no cache. The "clear" policy is the default.</li> </ul> |
| <code>BB_STAMP_POLICY</code>              | <p>Defines the mode used for how timestamps of stamp files are compared. You can set the variable to one of the following modes:</p> <ul style="list-style-type: none"> <li>• <b>perfile</b> - Timestamp comparisons are only made between timestamps of a specific recipe. This is the default mode.</li> <li>• <b>full</b> - Timestamp comparisons are made for all dependencies.</li> <li>• <b>whitelist</b> - Identical to "full" mode except timestamp comparisons are made for recipes listed in the <code>BB_STAMP_WHITELIST</code> variable.</li> </ul>                                      |

## Note

Stamp policies are largely obsolete with the introduction of setscene tasks.

**BB\_STAMP\_WHITELIST**

Lists files whose stamp file timestamps are compared when the stamp policy mode is set to "whitelist". For information on stamp policies, see the [BB\\_STAMP\\_POLICY](#) variable.

**BB\_STRICT\_CHECKSUM**

Sets a more strict checksum mechanism for non-local URLs. Setting this variable to a value causes BitBake to report an error if it encounters a non-local URL that does not have at least one checksum specified.

**BB\_TASK\_IONICE\_LEVEL**

Allows adjustment of a task's Input/Output priority. During Autobuilder testing, random failures can occur for tasks due to I/O starvation. These failures occur during various QEMU runtime timeouts. You can use the [BB\\_TASK\\_IONICE\\_LEVEL](#) variable to adjust the I/O priority of these tasks.

**Note**

This variable works similarly to the [BB\\_TASK\\_NICE\\_LEVEL](#) variable except with a task's I/O priorities.

Set the variable as follows:

```
BB_TASK_IONICE_LEVEL = "class.prio"
```

For *class*, the default value is "2", which is a best effort. You can use "1" for realtime and "3" for idle. If you want to use realtime, you must have superuser privileges.

For *prio*, you can use any value from "0", which is the highest priority, to "7", which is the lowest. The default value is "4". You do not need any special privileges to use this range of priority values.

**Note**

In order for your I/O priority settings to take effect, you need the Completely Fair Queuing (CFQ) Scheduler selected for the backing block device. To select the scheduler, use the following command form where *device* is the device (e.g. sda, sdb, and so forth):

```
$ sudo sh -c "echo cfq > /sys/block/device/queue/scheduler"
```

**BB\_TASK\_NICE\_LEVEL**

Allows specific tasks to change their priority (i.e. nice level).

You can use this variable in combination with task overrides to raise or lower priorities of specific tasks. For example, on the [Yocto Project](#) autobuilder, QEMU emulation in images is given a higher priority as compared to build tasks to ensure that images do not suffer timeouts on loaded systems.

**BB\_TASKHASH**

Within an executing task, this variable holds the hash of the task as returned by the currently enabled signature generator.

**BB\_VERBOSE\_LOGS**

Controls how verbose BitBake is during builds. If set, shell scripts echo commands and shell script output appears on standard out (stdout).

**BB\_WORKERCONTEXT**

Specifies if the current context is executing a task. BitBake sets this variable to "1" when a task is being executed. The value is not set when the task is in server context during parsing or event handling.

**BBCLASSEXTEND**

Allows you to extend a recipe so that it builds variants of the software. Some examples of these variants for recipes from the OpenEmbedded-Core metadata are "natives" such as `quilt-native`, which is a copy of Quilt built to run on the build system; "crosses" such as `gcc-cross`, which is a compiler built to run on the build machine but produces binaries that run on the target MACHINE; "nativesdk", which targets the SDK machine instead of MACHINE; and "multilibs" in the form `"multilib:multilib_name"`.

To build a different variant of the recipe with a minimal amount of code, it usually is as simple as adding the variable to your recipe. Here are two examples. The "native" variants are from the OpenEmbedded-Core metadata:

```
BBCLASSEXTEND += "native nativesdk"
BBCLASSEXTEND += "multilib:multilib_name"
```

**Note**

Internally, the `BBCLASSEXTEND` mechanism generates recipe variants by rewriting variable values and applying overrides such as `_class-native`. For example, to generate a native version of a recipe, a `DEPENDS` on "foo" is rewritten to a `DEPENDS` on "foo-native".

Even when using `BBCLASSEXTEND`, the recipe is only parsed once. Parsing once adds some limitations. For example, it is not possible to include a different file depending on the variant, since `include` statements are processed when the recipe is parsed.

**BBDEBUG**

Sets the BitBake debug output level to a specific value as incremented by the `-D` command line option.

**Note**

You must set this variable in the external environment in order for it to work.

**BBFILE\_COLLECTIONS**

Lists the names of configured layers. These names are used to find the other `BBFILE_*` variables. Typically, each layer appends its name to this variable in its `conf/layer.conf` file.

**BBFILE\_PATTERN**

Variable that expands to match files from `BBFILES` in a particular layer. This variable is used in the `conf/layer.conf` file and must be suffixed with the name of the specific layer (e.g. `BBFILE_PATTERN_emenlow`).

**BBFILE\_PRIORITY**

Assigns the priority for recipe files in each layer.

This variable is useful in situations where the same recipe appears in more than one layer. Setting this variable allows you to prioritize a layer against other layers that contain the same recipe - effectively letting you control the precedence for the multiple layers. The precedence established through this variable stands regardless of a recipe's version (`PV` variable). For example, a layer that has a recipe with a higher `PV` value but for which the `BBFILE_PRIORITY` is set to have a lower precedence still has a lower precedence.

A larger value for the `BBFILE_PRIORITY` variable results in a higher precedence. For example, the value 6 has a higher precedence than the value 5. If not specified, the `BBFILE_PRIORITY` variable is set based on layer dependencies (see the `LAYERDEPENDS` variable for more information. The default priority, if unspecified for a layer with no dependencies, is the lowest defined priority + 1 (or 1 if no priorities are defined).

**Tip**

You can use the command `bitbake-layers show-layers` to list all configured layers along with their priorities.

**BBFILES**

A space-separated list of recipe files BitBake uses to build software.

When specifying recipe files, you can pattern match using Python's `glob` syntax. For details on the syntax, see the documentation by following the previous link.

**BBINCLUDED**

Contains a space-separated list of all of all files that BitBake's parser included during parsing of the current file.

**BBINCLUDELOGS**

If set to a value, enables printing the task log when reporting a failed task.

**BBINCLUDELOGS\_LINES**

If `BBINCLUDELOGS` is set, specifies the maximum number of lines from the task log file to print when reporting a failed task. If you do not set `BBINCLUDELOGS_LINES`, the entire log is printed.

**BBLAYERS**

Lists the layers to enable during the build. This variable is defined in the `bblayers.conf` configuration file in the build directory. Here is an example:

```
BBLAYERS = " \
/home/scottrif/poky/meta \
/home/scottrif/poky/meta-yocto \
/home/scottrif/poky/meta-yocto-bsp \
/home/scottrif/poky/meta-mykernel \
"
```

This example enables four layers, one of which is a custom, user-defined layer named `meta-mykernel`.

**BBLAYERS\_FETCH\_DIR**



|                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                      | <p>Sets the base location where layers are stored. This setting is used in conjunction with <code>bitbake-layers layerindex-fetch</code> and tells <code>bitbake-layers</code> where to place the fetched layers.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>BBMASK</b>        | <p>Prevents BitBake from processing recipes and recipe append files.</p> <p>You can use the <code>BBMASK</code> variable to "hide" these <code>.bb</code> and <code>.bbappend</code> files. BitBake ignores any recipe or recipe append files that match any of the expressions. It is as if BitBake does not see them at all. Consequently, matching files are not parsed or otherwise used by BitBake.</p> <p>The values you provide are passed to Python's regular expression compiler. Consequently, the syntax follows Python's Regular Expression (re) syntax. The expressions are compared against the full paths to the files. For complete syntax information, see Python's documentation at <a href="http://docs.python.org/3/library/re.html#re">http://docs.python.org/3/library/re.html#re</a>.</p> <p>The following example uses a complete regular expression to tell BitBake to ignore all recipe and recipe append files in the <code>meta-ti/recipes-misc/</code> directory:</p> <pre>BBMASK = "meta-ti/recipes-misc/"</pre> <p>If you want to mask out multiple directories or recipes, you can specify multiple regular expression fragments. This next example masks out multiple directories and individual recipes:</p> <pre>BBMASK += "/meta-ti/recipes-misc/ meta-ti/recipes-ti/packagegroup/" BBMASK += "/meta-oe/recipes-support/" BBMASK += "/meta-foo/.*openldap" BBMASK += "opencv.*\bbappend" BBMASK += "lzma"</pre> |
|                      | <p><b>Note</b></p> <p>When specifying a directory name, use the trailing slash character to ensure you match just that directory name.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>BBMULTICONFIG</b> | <p>Enables BitBake to perform multiple configuration builds and lists each separate configuration (multiconfig). You can use this variable to cause BitBake to build multiple targets where each target has a separate configuration. Define <code>BBMULTICONFIG</code> in your <code>conf/local.conf</code> configuration file.</p> <p>As an example, the following line specifies three multiconfigs, each having a separate configuration file:</p> <pre>BBMULTIFONFIG = "configA configB configC"</pre> <p>Each configuration file you use must reside in the build directory within a directory named <code>conf/multiconfig</code> (e.g. <code>build_directory/conf/multiconfig/configA.conf</code>).</p> <p>For information on how to use <code>BBMULTICONFIG</code> in an environment that supports building targets with multiple configurations, see the "<a href="#">Executing a Multiple Configuration Build</a>" section.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>BBPATH</b>        | <p>Used by BitBake to locate class (<code>.bbclass</code>) and configuration (<code>.conf</code>) files. This variable is analogous to the <code>PATH</code> variable.</p> <p>If you run BitBake from a directory outside of the build directory, you must be sure to set <code>BBPATH</code> to point to the build directory. Set the variable as you would any environment variable and then run BitBake:</p> <pre>\$ BBPATH="build_directory" \$ export BBPATH \$ bitbake target</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>BBSERVER</b>      | <p>Points to the server that runs memory-resident BitBake. The variable is only used when you employ memory-resident BitBake.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>BBTARGETS</b>     | <p>Allows you to use a configuration file to add to the list of command-line target recipes you want to build.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>BBVERSIONS</b>    | <p>Allows a single recipe to build multiple versions of a project from a single recipe file. You also able to specify conditional metadata using the <a href="#">OVERRIDES</a> mechanism for a single version or for an optionally named range of versions.</p> <p>For more information on <code>BBVERSIONS</code>, see the "<a href="#">Variants - Class Extension Mechanism</a>" section.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>BITBAKE_UI</b>    | <p>Used to specify the UI module to use when running BitBake. Using this variable is equivalent to using the <code>-u</code> command-line option.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |

## Note

You must set this variable in the external environment in order for it to work.

**BUILDNAME**¶ A name assigned to the build. The name defaults to a datetime stamp of when the build was started but can be defined by the metadata.

**BZRDIR**¶ The directory in which files checked out of a Bazaar system are stored.

## C

**CACHE**¶ Specifies the directory BitBake uses to store a cache of the metadata so it does not need to be parsed every time BitBake is started.

**CVSDIR**¶ The directory in which files checked out under the CVS system are stored.

## D

**DEFAULT\_PREFERENCE**¶ Specifies a weak bias for recipe selection priority.

The most common usage of this variable is to set it to "-1" within a recipe for a development version of a piece of software. Using the variable in this way causes the stable version of the recipe to build by default in the absence of PREFERRED\_VERSION being used to build the development version.

### Note

The bias provided by `DEFAULT_PREFERENCE` is weak and is overridden by BBFILE\_PRIORITY if that variable is different between two layers that contain different versions of the same recipe.

**DEPENDS**¶ Lists a recipe's build-time dependencies (i.e. other recipe files).

Consider this simple example for two recipes named "a" and "b" that produce similarly named packages. In this example, the `DEPENDS` statement appears in the "a" recipe:

```
DEPENDS = "b"
```

Here, the dependency is such that the `do_configure` task for recipe "a" depends on the `do_populate_sysroot` task of recipe "b". This means anything that recipe "b" puts into sysroot is available when recipe "a" is configuring itself.

For information on runtime dependencies, see the RDEPENDS variable.

**DESCRIPTION**¶ A long description for the recipe.

**DL\_DIR**¶ The central download directory used by the build process to store downloads. By default, `DL_DIR` gets files suitable for mirroring for everything except Git repositories. If you want tarballs of Git repositories, use the BB\_GENERATE\_MIRROR\_TARBALLS variable.

## E

**EXCLUDE\_FROM\_WORLD**¶ Directs BitBake to exclude a recipe from world builds (i.e. `bitbake world`). During world builds, BitBake locates, parses and builds all recipes found in every layer exposed in the `bblayers.conf` configuration file.

To exclude a recipe from a world build using this variable, set the variable to "1" in the recipe.

### Note

Recipes added to `EXCLUDE_FROM_WORLD` may still be built during a world build in order to satisfy dependencies of other recipes. Adding a recipe to `EXCLUDE_FROM_WORLD` only ensures that the recipe is not explicitly added to the list of build targets in a world build.

## F

**FAKEROOT**¶ Contains the command to use when running a shell script in a fakeroot environment. The `FAKEROOT` variable is obsolete and has been replaced by the other `FAKEROOT*` variables. See these entries in the glossary for more information.

**FAKEROOTBASEENV**¶

Lists environment variables to set when executing the command defined by [FAKEROOTCMD](#) that starts the bitbake-worker process in the fakeroot environment.

**FAKEROOTCMD**¶ Contains the command that starts the bitbake-worker process in the fakeroot environment.

**FAKEROOTDIRS**¶ Lists directories to create before running a task in the fakeroot environment.

**FAKEROOTENV**¶ Lists environment variables to set when running a task in the fakeroot environment. For additional information on environment variables and the fakeroot environment, see the [FAKEROOTBASEENV](#) variable.

**FAKEROOTNOENV**¶ Lists environment variables to set when running a task that is not in the fakeroot environment. For additional information on environment variables and the fakeroot environment, see the [FAKEROOTENV](#) variable.

**FETCHCMD**¶ Defines the command the BitBake fetcher module executes when running fetch operations. You need to use an override suffix when you use the variable (e.g. `FETCHCMD_git` or `FETCHCMD_svn`).

**FILE**¶ Points at the current file. BitBake sets this variable during the parsing process to identify the file being parsed. BitBake also sets this variable when a recipe is being executed to identify the recipe file.

**FILESPATH**¶ Specifies directories BitBake uses when searching for patches and files. The "local" fetcher module uses these directories when handling `file://` URLs. The variable behaves like a shell `PATH` environment variable. The value is a colon-separated list of directories that are searched left-to-right in order.

## G

**GITDIR**¶ The directory in which a local copy of a Git repository is stored when it is cloned.

## H

**HGDIR**¶ The directory in which files checked out of a Mercurial system are stored.

**HOMEPAGE**¶ Website where more information about the software the recipe is building can be found.

## I

**INHERIT**¶ Causes the named class or classes to be inherited globally. Anonymous functions in the class or classes are not executed for the base configuration and in each individual recipe. The OpenEmbedded build system ignores changes to `INHERIT` in individual recipes.

For more information on `INHERIT`, see the "[INHERIT Configuration Directive](#)" section.

## L

**LAYERDEPENDS**¶ Lists the layers, separated by spaces, upon which this recipe depends. Optionally, you can specify a specific layer version for a dependency by adding it to the end of the layer name with a colon, (e.g. "anotherlayer:3" to be compared against [LAYERVERSION](#) anotherlayer in this case). BitBake produces an error if any dependency is missing or the version numbers do not match exactly (if specified).

You use this variable in the `conf/layer.conf` file. You must also use the specific layer name as a suffix to the variable (e.g. `LAYERDEPENDS_mylayer`).

**LAYERDIR**¶ When used inside the `layer.conf` configuration file, this variable provides the path of the current layer. This variable is not available outside of `layer.conf` and references are expanded immediately when parsing of the file completes.

**LAYERDIR\_RE**¶ When used inside the `layer.conf` configuration file, this variable provides the path of the current layer, escaped for use in a regular expression ([BBFILE\\_PATTERN](#)). This variable is not available outside of `layer.conf` and references are expanded immediately when parsing of the file completes.

**LAYERVERSION**¶ Optionally specifies the version of a layer as a single number. You can use this variable within [LAYERDEPENDS](#) for another layer in order to depend on a specific version of the layer.

You use this variable in the `conf/layer.conf` file. You must also use the specific layer name as a suffix to the variable (e.g. `LAYERDEPENDS_mylayer`).

**LICENSE**¶ The list of source licenses for the recipe.

## M

## O

**MIRRORS**

Specifies additional paths from which BitBake gets source code. When the build system searches for source code, it first tries the local download directory. If that location fails, the build system tries locations defined by [PREMIRRORS](#), the upstream source, and then locations specified by **MIRRORS** in that order.

**MULTI\_PROVIDER\_WHITELIST**

Allows you to suppress BitBake warnings caused when building two separate recipes that provide the same output.

Bitbake normally issues a warning when building two different recipes where each provides the same output. This scenario is usually something the user does not want. However, cases do exist where it makes sense, particularly in the `virtual/*` namespace. You can use this variable to suppress BitBake's warnings.

To use the variable, list provider names (e.g. recipe names, `virtual/kernel`, and so forth).

**OVERRIDES**

BitBake uses **OVERRIDES** to control what variables are overridden after BitBake parses recipes and configuration files.

Following is a simple example that uses an overrides list based on machine architectures:

```
OVERRIDES = "arm:x86:mips:powerpc"
```

You can find information on how to use **OVERRIDES** in the "[Conditional Syntax \(Overrides\)](#)" section.

## P

**P4DIR**

The directory in which a local copy of a Perforce depot is stored when it is fetched.

**PACKAGES**

The list of packages the recipe creates.

**PACKAGES\_DYNAMIC**

A promise that your recipe satisfies runtime dependencies for optional modules that are found in other recipes. **PACKAGES\_DYNAMIC** does not actually satisfy the dependencies, it only states that they should be satisfied. For example, if a hard, runtime dependency ([RDEPENDS](#)) of another package is satisfied during the build through the **PACKAGES\_DYNAMIC** variable, but a package with the module name is never actually produced, then the other package will be broken.

**PE**

The epoch of the recipe. By default, this variable is unset. The variable is used to make upgrades possible when the versioning scheme changes in some backwards incompatible way.

**PERSISTENT\_DIR**

Specifies the directory BitBake uses to store data that should be preserved between builds. In particular, the data stored is the data that uses BitBake's persistent data API and the data used by the PR Server and PR Service.

**PF**

Specifies the recipe or package name and includes all version and revision numbers (i.e. `eglibc-2.13-r20+svnr15508/` and `bash-4.2-r1/`).

**PN**

The recipe name.

**PR**

The revision of the recipe.

**PREFERRED\_PROVIDER**

Determines which recipe should be given preference when multiple recipes provide the same item. You should always suffix the variable with the name of the provided item, and you should set it to the [PN](#) of the recipe to which you want to give precedence. Some examples:

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
PREFERRED_PROVIDER_virtual/xserver = "xserver-xf86"
PREFERRED_PROVIDER_virtual/libgl ?= "mesa"
```

**PREFERRED\_PROVIDERS**

Determines which recipe should be given preference for cases where multiple recipes provide the same item. Functionally, **PREFERRED\_PROVIDERS** is identical to [PREFERRED\\_PROVIDER](#). However, the **PREFERRED\_PROVIDERS** variable lets you define preferences for multiple situations using the following form:

```
PREFERRED_PROVIDERS = "xxx:yyy aaa:bbb ..."
```

This form is a convenient replacement for the following:

```
PREFERRED_PROVIDER_xxx = "yyy"
PREFERRED_PROVIDER_aaa = "bbb"
```

**PREFERRED\_VERSION**

If there are multiple versions of recipes available, this variable determines which recipe should be given preference. You must always suffix the variable with the [PN](#) you want to select, and you should set [PV](#) accordingly for precedence.

The **PREFERRED\_VERSION** variable supports limited wildcard use through the `%` character. You can use the character to match any number of characters, which can be useful when specifying versions that contain long revision numbers that potentially change. Here are two examples:

```
PREFERRED_VERSION_python = "2.7.3"
PREFERRED_VERSION_linux-yocto = "4.12"
```

## Important

The use of the "%" character is limited in that it only works at the end of the string. You cannot use the wildcard character in any other location of the string.

### PREMIRRORS¶

Specifies additional paths from which BitBake gets source code. When the build system searches for source code, it first tries the local download directory. If that location fails, the build system tries locations defined by PREMIRRORS, the upstream source, and then locations specified by [MIRRORS](#) in that order.

Typically, you would add a specific server for the build system to attempt before any others by adding something like the following to your configuration:

```
PREMIRRORS_prepend = "\
git://.*.* http://www.yoctoproject.org/sources/ \n \
ftp://.*.* http://www.yoctoproject.org/sources/ \n \
http://.*.* http://www.yoctoproject.org/sources/ \n \
https://.*.* http://www.yoctoproject.org/sources/ \n"
```

These changes cause the build system to intercept Git, FTP, HTTP, and HTTPS requests and direct them to the `http://sources.mirror`. You can use `file://` URLs to point to local directories or network shares as well.

### PROVIDES¶

A list of aliases by which a particular recipe can be known. By default, a recipe's own [PN](#) is implicitly already in its PROVIDES list. If a recipe uses PROVIDES, the additional aliases are synonyms for the recipe and can be useful satisfying dependencies of other recipes during the build as specified by [DEPENDS](#).

Consider the following example PROVIDES statement from a recipe file `libav_0.8.11.bb`:

```
PROVIDES += "libpostproc"
```

The PROVIDES statement results in the "libav" recipe also being known as "libpostproc".

In addition to providing recipes under alternate names, the PROVIDES mechanism is also used to implement virtual targets. A virtual target is a name that corresponds to some particular functionality (e.g. a Linux kernel). Recipes that provide the functionality in question list the virtual target in PROVIDES. Recipes that depend on the functionality in question can include the virtual target in [DEPENDS](#) to leave the choice of provider open.

Conventionally, virtual targets have names on the form "virtual/function" (e.g. "virtual/kernel"). The slash is simply part of the name and has no syntactical significance.

### PRSERV\_HOST¶

The network based [PR](#) service host and port.

Following is an example of how the PRSERV\_HOST variable is set:

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

You must set the variable if you want to automatically start a local PR service. You can set PRSERV\_HOST to other values to use a remote PR service.

### PV¶

The version of the recipe.

## R

### RDEPENDS¶

Lists a package's runtime dependencies (i.e. other packages) that must be installed in order for the built package to run correctly. If a package in this list cannot be found during the build, you will get a build error.

Because the RDEPENDS variable applies to packages being built, you should always use the variable in a form with an attached package name. For example, suppose you are building a development package that depends on the `perl` package. In this case, you would use the following RDEPENDS statement:

```
RDEPENDS_${PN}-dev += "perl"
```

In the example, the development package depends on the `perl` package. Thus, the RDEPENDS variable has the `${PN}-dev` package name as part of the variable.

BitBake supports specifying versioned dependencies. Although the syntax varies depending on the packaging format, BitBake hides these differences from you. Here is the general syntax to specify versions with the RDEPENDS variable:

```
RDEPENDS_${PN} = "package (operator version)"
```

For `operator`, you can specify the following:

```
=
<
>
<=
>=
```

For example, the following sets up a dependency on version 1.2 or greater of the package `foo`:

```
RDEPENDS_${PN} = "foo (>= 1.2)"
```

For information on build-time dependencies, see the [DEPENDS](#) variable.

#### REPODIR¶

The directory in which a local copy of a `google-repo` directory is stored when it is synced.

#### RPROVIDES¶

A list of package name aliases that a package also provides. These aliases are useful for satisfying runtime dependencies of other packages both during the build and on the target (as specified by [RDEPENDS](#)).

As with all package-controlling variables, you must always use the variable in conjunction with a package name override. Here is an example:

```
RPROVIDES_${PN} = "widget-abi-2"
```

#### RRECOMMENDS¶

A list of packages that extends the usability of a package being built. The package being built does not depend on this list of packages in order to successfully build, but needs them for the extended usability. To specify runtime dependencies for packages, see the [RDEPENDS](#) variable.

BitBake supports specifying versioned recommends. Although the syntax varies depending on the packaging format, BitBake hides these differences from you. Here is the general syntax to specify versions with the `RRECOMMENDS` variable:

```
RRECOMMENDS_${PN} = "package (operator version)"
```

For `operator`, you can specify the following:

```
=
<
>
<=
>=
```

For example, the following sets up a recommend on version 1.2 or greater of the package `foo`:

```
RRECOMMENDS_${PN} = "foo (>= 1.2)"
```

## S

#### SECTION¶

The section in which packages should be categorized.

#### SRC\_URI¶

The list of source files - local or remote. This variable tells BitBake which bits to pull for the build and how to pull them. For example, if the recipe or append file needs to fetch a single tarball from the Internet, the recipe or append file uses a `SRC_URI` entry that specifies that tarball. On the other hand, if the recipe or append file needs to fetch a tarball and include a custom file, the recipe or append file needs an `SRC_URI` variable that specifies all those sources.

The following list explains the available URI protocols:

- **file://** - Fetches files, which are usually files shipped with the metadata, from the local machine. The path is relative to the [FILES\\_PATH](#) variable.
- **bzr://** - Fetches files from a Bazaar revision control repository.
- **git://** - Fetches files from a Git revision control repository.
- **osc://** - Fetches files from an OSC (OpenSUSE Build service) revision control repository.
- **repo://** - Fetches files from a repo (Git) repository.
- **http://** - Fetches files from the Internet using HTTP.
- **https://** - Fetches files from the Internet using HTTPS.
- **ftp://** - Fetches files from the Internet using FTP.
- **cvs://** - Fetches files from a CVS revision control repository.
- **hg://** - Fetches files from a Mercurial (`hg`) revision control repository.
- **p4://** - Fetches files from a Perforce (`p4`) revision control repository.
- **ssh://** - Fetches files from a secure shell.

- ***svn://*** - Fetches files from a Subversion (`svn`) revision control repository.

Here are some additional options worth mentioning:

- ***unpack*** - Controls whether or not to unpack the file if it is an archive. The default action is to unpack the file.
- ***subdir*** - Places the file (or extracts its contents) into the specified subdirectory. This option is useful for unusual tarballs or other archives that do not have their files already in a subdirectory within the archive.
- ***name*** - Specifies a name to be used for association with `SRC_URI` checksums when you have more than one file specified in `SRC_URI`.
- ***downloadfilename*** - Specifies the filename used when storing the downloaded file.

`SRCDATE`

The date of the source code used to build the package. This variable applies only if the source was fetched from a Source Code Manager (SCM).

`SRCREV`

The revision of the source code used to build the package. This variable applies only when using Subversion, Git, Mercurial and Bazaar. If you want to build a fixed revision and you want to avoid performing a query on the remote repository every time BitBake parses your recipe, you should specify a `SRCREV` that is a full revision identifier and not just a tag.

`SRCREV_FORMAT`

Helps construct valid `SRCREV` values when multiple source controlled URLs are used in `SRC_URI`.

The system needs help constructing these values under these circumstances. Each component in the `SRC_URI` is assigned a name and these are referenced in the `SRCREV_FORMAT` variable. Consider an example with URLs named "machine" and "meta". In this case, `SRCREV_FORMAT` could look like "machine\_meta" and those names would have the SCM versions substituted into each position. Only one `AUTOINC` placeholder is added and if needed. And, this placeholder is placed at the start of the returned string.

`STAMP`

Specifies the base path used to create recipe stamp files. The path to an actual stamp file is constructed by evaluating this string and then appending additional information.

`STAMPCLEAN`

Specifies the base path used to create recipe stamp files. Unlike the `STAMP` variable, `STAMPCLEAN` can contain wildcards to match the range of files a clean operation should remove. BitBake uses a clean operation to remove any other stamps it should be removing when creating a new stamp.

`SUMMARY`

A short summary for the recipe, which is 72 characters or less.

`SVNDIR`

The directory in which files checked out of a Subversion system are stored.

## T

`T`

Points to a directory where BitBake places temporary files, which consist mostly of task logs and scripts, when building a particular recipe.

`TOPDIR`

Points to the build directory. BitBake automatically sets this variable.

## Appendix F. Hello World Example

[F.1. BitBake Hello World](#)

[F.2. Obtaining BitBake](#)

[F.3. Setting Up the BitBake Environment](#)

[F.4. The Hello World Example](#)

### F.1. BitBake Hello World

The simplest example commonly used to demonstrate any new programming language or tool is the "Hello World" example. This appendix demonstrates, in tutorial form, Hello World within the context of BitBake. The tutorial describes how to create a new project and the applicable metadata files necessary to allow BitBake to build it.

### F.2. Obtaining BitBake

See the "Obtaining BitBake" section for information on how to obtain BitBake. Once you have the source code on your machine, the BitBake directory appears as follows:

```
$ ls -al
total 100
drwxrwxr-x. 9 wmat wmat 4096 Jan 31 13:44 .
drwxrwxr-x. 3 wmat wmat 4096 Feb 4 10:45 ..
-rw-rw-r--. 1 wmat wmat 365 Nov 26 04:55 AUTHORS
drwxrwxr-x. 2 wmat wmat 4096 Nov 26 04:55 bin
drwxrwxr-x. 4 wmat wmat 4096 Jan 31 13:44 build
-rw-rw-r--. 1 wmat wmat 16501 Nov 26 04:55 ChangeLog
drwxrwxr-x. 2 wmat wmat 4096 Nov 26 04:55 classes
```

```
drwxrwxr-x. 2 wmat wmat 4096 Nov 26 04:55 conf
drwxrwxr-x. 3 wmat wmat 4096 Nov 26 04:55 contrib
-rw-rw-r--. 1 wmat wmat 17987 Nov 26 04:55 COPYING
drwxrwxr-x. 3 wmat wmat 4096 Nov 26 04:55 doc
-rw-rw-r--. 1 wmat wmat 69 Nov 26 04:55 .gitignore
-rw-rw-r--. 1 wmat wmat 849 Nov 26 04:55 HEADER
drwxrwxr-x. 5 wmat wmat 4096 Jan 31 13:44 lib
-rw-rw-r--. 1 wmat wmat 195 Nov 26 04:55 MANIFEST.in
-rw-rw-r--. 1 wmat wmat 2887 Nov 26 04:55 TODO
```

At this point, you should have BitBake cloned to a directory that matches the previous listing except for dates and user names.

### F.3. Setting Up the BitBake Environment¶

First, you need to be sure that you can run BitBake. Set your working directory to where your local BitBake files are and run the following command:

```
$./bin/bitbake --version
BitBake Build Tool Core version 1.23.0, bitbake version 1.23.0
```

The console output tells you what version you are running.

The recommended method to run BitBake is from a directory of your choice. To be able to run BitBake from any directory, you need to add the executable binary to your binary to your shell's environment PATH variable. First, look at your current PATH variable by entering the following:

```
$ echo $PATH
```

Next, add the directory location for the BitBake binary to the PATH. Here is an example that adds the `/home/scott-lenovo/bitbake/bin` directory to the front of the PATH variable:

```
$ export PATH=/home/scott-lenovo/bitbake/bin:$PATH
```

You should now be able to enter the `bitbake` command from the command line while working from any directory.

### F.4. The Hello World Example¶

The overall goal of this exercise is to build a complete "Hello World" example utilizing task and layer concepts. Because this is how modern projects such as OpenEmbedded and the Yocto Project utilize BitBake, the example provides an excellent starting point for understanding BitBake.

To help you understand how to use BitBake to build targets, the example starts with nothing but the `bitbake` command, which causes BitBake to fail and report problems. The example progresses by adding pieces to the build to eventually conclude with a working, minimal "Hello World" example.

While every attempt is made to explain what is happening during the example, the descriptions cannot cover everything. You can find further information throughout this manual. Also, you can actively participate in the <http://lists.openembedded.org/mailman/listinfo/bitbake-devel> discussion mailing list about the BitBake build tool.

#### Note

This example was inspired by and drew heavily from [Mailing List post - The BitBake equivalent of "Hello, World!"](#).

As stated earlier, the goal of this example is to eventually compile "Hello World". However, it is unknown what BitBake needs and what you have to provide in order to achieve that goal. Recall that BitBake utilizes three types of metadata files: [Configuration Files](#), [Classes](#), and [Recipes](#). But where do they go? How does BitBake find them? BitBake's error messaging helps you answer these types of questions and helps you better understand exactly what is going on.

Following is the complete "Hello World" example.

1. **Create a Project Directory:** First, set up a directory for the "Hello World" project. Here is how you can do so in your home directory:

```
$ mkdir ~/hello
$ cd ~/hello
```

This is the directory that BitBake will use to do all of its work. You can use this directory to keep all the metafiles needed by BitBake. Having a project directory is a good way to isolate your project.

2. **Run Bitbake:** At this point, you have nothing but a project directory. Run the `bitbake` command and see what it does:

```
$ bitbake
The BBPATH variable is not set and bitbake did not
find a conf/bblayers.conf file in the expected location.
Maybe you accidentally invoked bitbake from the wrong directory?
DEBUG: Removed the following variables from the environment:
GNOME_DESKTOP_SESSION_ID, XDG_CURRENT_DESKTOP,
GNOME_KEYRING_CONTROL, DISPLAY, SSH_AGENT_PID, LANG, no_proxy,
XDG_SESSION_PATH, XAUTHORITY, SESSION_MANAGER, SHLV,
MANDATORY_PATH, COMPIZ_CONFIG_PROFILE, WINDOWID, EDITOR,
GPG_AGENT_INFO, SSH_AUTH_SOCK, GDMSESSION, GNOME_KEYRING_PID,
XDG_SEAT_PATH, XDG_CONFIG_DIRS, LESSOPEN, DBUS_SESSION_BUS_ADDRESS,
_, XDG_SESSION_COOKIE, DESKTOP_SESSION, LESSCLOSE, DEFAULTS_PATH,
UBUNTU_MENUPROXY, OLDPWD, XDG_DATA_DIRS, COLORTERM, LS_COLORS
```



The majority of this output is specific to environment variables that are not directly relevant to BitBake. However, the very first message regarding the `BBPATH` variable and the `conf/bblayers.conf` file is relevant.

When you run BitBake, it begins looking for metadata files. The `BBPATH` variable is what tells BitBake where to look for those files. `BBPATH` is not set and you need to set it. Without `BBPATH`, BitBake cannot find any configuration files (`.conf`) or recipe files (`.bb`) at all. BitBake also cannot find the `bitbake.conf` file.

3. **Setting `BBPATH`:** For this example, you can set `BBPATH` in the same manner that you set `PATH` earlier in the appendix. You should realize, though, that it is much more flexible to set the `BBPATH` variable up in a configuration file for each project.

From your shell, enter the following commands to set and export the `BBPATH` variable:

```
$ BBPATH="projectdirectory"
$ export BBPATH
```

Use your actual project directory in the command. BitBake uses that directory to find the metadata it needs for your project.

## Note

When specifying your project directory, do not use the tilde ("`~`") character as BitBake does not expand that character as the shell would.

4. **Run `BitBake`:** Now that you have `BBPATH` defined, run the `bitbake` command again:

```
$ bitbake
ERROR: Traceback (most recent call last):
 File "/home/scott-lenovo/bitbake/lib/bb/cookerdata.py", line 163, in wrapped
 return func(fn, *args)
 File "/home/scott-lenovo/bitbake/lib/bb/cookerdata.py", line 173, in parse_config_file
 return bb.parse.handle(fn, data, include)
 File "/home/scott-lenovo/bitbake/lib/bb/parse/__init__.py", line 99, in handle
 return h['handle'](fn, data, include)
 File "/home/scott-lenovo/bitbake/lib/bb/parse/parse_py/ConfHandler.py", line 120, in handle
 abs_fn = resolve_file(fn, data)
 File "/home/scott-lenovo/bitbake/lib/bb/parse/__init__.py", line 117, in resolve_file
 raise IOError("file %s not found in %s" % (fn, bbpath))
IOError: file conf/bitbake.conf not found in /home/scott-lenovo/hello

ERROR: Unable to parse conf/bitbake.conf: file conf/bitbake.conf not found in /home/scott-lenovo/hello
```

This sample output shows that BitBake could not find the `conf/bitbake.conf` file in the project directory. This file is the first thing BitBake must find in order to build a target. And, since the project directory for this example is empty, you need to provide a `conf/bitbake.conf` file.

5. **Creating `conf/bitbake.conf`:** The `conf/bitbake.conf` includes a number of configuration variables BitBake uses for metadata and recipe files. For this example, you need to create the file in your project directory and define some key BitBake variables. For more information on the `bitbake.conf` file, see <http://git.openembedded.org/bitbake/tree/conf/bitbake.conf>.

Use the following commands to create the `conf` directory in the project directory:

```
$ mkdir conf
```

From within the `conf` directory, use some editor to create the `bitbake.conf` so that it contains the following:

```
PN = "${@bb.parse.BBHandler.vars_from_file(d.getVar('FILE', False),d)[0] or 'defaultpkgname'}"

TMPDIR = "${TOPDIR}/tmp"
CACHE = "${TMPDIR}/cache"
STAMP = "${TMPDIR}/${PN}/stamps"
T = "${TMPDIR}/${PN}/work"
B = "${TMPDIR}/${PN}"
```

## Note

Without a value for `PN`, the variables `STAMP`, `T`, and `B`, prevent more than one recipe from working. You can fix this by either setting `PN` to have a value similar to what OpenEmbedded and BitBake use in the default `bitbake.conf` file (see previous example). Or, by manually updating each recipe to set `PN`. You will also need to include `PN` as part of the `STAMP`, `T`, and `B` variable definitions in the `local.conf` file.

The `TMPDIR` variable establishes a directory that BitBake uses for build output and intermediate files other than the cached information used by the [Setscene](#) process. Here, the `TMPDIR` directory is set to `hello/tmp`.

## Tip

You can always safely delete the `tmp` directory in order to rebuild a BitBake target. The build process creates the directory for you when you run BitBake.

For information about each of the other variables defined in this example, click on the links to take you to the definitions in the glossary.

6. **Run Bitbake:** After making sure that the `conf/bitbake.conf` file exists, you can run the `bitbake` command again:

```
$ bitbake
ERROR: Traceback (most recent call last):
 File "/home/scott-lenovo/bitbake/lib/bb/cookerdata.py", line 163, in wrapped
 return func(fn, *args)
 File "/home/scott-lenovo/bitbake/lib/bb/cookerdata.py", line 177, in _inherit
 bb.parse.BBHandler.inherit(bbclass, "configuration INHERITS", 0, data)
 File "/home/scott-lenovo/bitbake/lib/bb/parse/parse_py/BBHandler.py", line 92, in inherit
 include(fn, file, lineno, d, "inherit")
 File "/home/scott-lenovo/bitbake/lib/bb/parse/parse_py/ConfHandler.py", line 100, in include
 raise ParseError("Could not %(error_out)s file %(fn)s" % vars(), oldfn, lineno)
ParseError: ParseError in configuration INHERITS: Could not inherit file classes/base.bbclass

ERROR: Unable to parse base: ParseError in configuration INHERITS: Could not inherit file classes/base.bbclass
```

In the sample output, BitBake could not find the `classes/base.bbclass` file. You need to create that file next.

7. **Creating classes/base.bbclass:** BitBake uses class files to provide common code and functionality. The minimally required class for BitBake is the `classes/base.bbclass` file. The `base` class is implicitly inherited by every recipe. BitBake looks for the class in the `classes` directory of the project (i.e `hello/classes` in this example).

Create the `classes` directory as follows:

```
$ cd $HOME/hello
$ mkdir classes
```

Move to the `classes` directory and then create the `base.bbclass` file by inserting this single line:

```
addtask build
```

The minimal task that BitBake runs is the `do_build` task. This is all the example needs in order to build the project. Of course, the `base.bbclass` can have much more depending on which build environments BitBake is supporting.

8. **Run Bitbake:** After making sure that the `classes/base.bbclass` file exists, you can run the `bitbake` command again:

```
$ bitbake
Nothing to do. Use 'bitbake world' to build everything, or run 'bitbake --help' for usage information.
```

BitBake is finally reporting no errors. However, you can see that it really does not have anything to do. You need to create a recipe that gives BitBake something to do.

9. **Creating a Layer:** While it is not really necessary for such a small example, it is good practice to create a layer in which to keep your code separate from the general metadata used by BitBake. Thus, this example creates and uses a layer called "mylayer".

## Note

You can find additional information on layers in the "[Layers](#)" section.

Minimally, you need a recipe file and a layer configuration file in your layer. The configuration file needs to be in the `conf` directory inside the layer. Use these commands to set up the layer and the `conf` directory:

```
$ cd $HOME
$ mkdir mylayer
$ cd mylayer
$ mkdir conf
```

Move to the `conf` directory and create a `layer.conf` file that has the following:

```
BBPATH .= ":${LAYERDIR}"

BBFILES += "${LAYERDIR}/*.bb"

BBFILE_COLLECTIONS += "mylayer"
BBFILE_PATTERN_mylayer := "^${LAYERDIR_RE}/"
```

For information on these variables, click the links to go to the definitions in the glossary.

You need to create the recipe file next. Inside your layer at the top-level, use an editor and create a recipe file named `printhello.bb` that has the following:

```
DESCRIPTION = "Prints Hello World"
PN = 'printhello'
PV = '1'

python do_build() {
 bb.plain("*****");
 bb.plain("*");
 bb.plain("* Hello, World! *");
 bb.plain("*");
 bb.plain("*****");
}
```

The recipe file simply provides a description of the recipe, the name, version, and the `do_build` task, which prints out "Hello World" to the console. For more information on these variables, follow the links to the glossary.

10. **Run Bitbake With a Target:** Now that a BitBake target exists, run the command and provide that target:

```
$ cd $HOME/hello
$ bitbake printhello
ERROR: no recipe files to build, check your BBPATH and BBFILES?

Summary: There was 1 ERROR message shown, returning a non-zero exit code.
```

We have created the layer with the recipe and the layer configuration file but it still seems that BitBake cannot find the recipe. BitBake needs a `conf/bblayers.conf` that lists the layers for the project. Without this file, BitBake cannot find the recipe.

11. **Creating `conf/bblayers.conf`:** BitBake uses the `conf/bblayers.conf` file to locate layers needed for the project. This file must reside in the `conf` directory of the project (i.e. `hello/conf` for this example).

Set your working directory to the `hello/conf` directory and then create the `bblayers.conf` file so that it contains the following:

```
BBLAYERS ?= " \
/home/<you>/mylayer \
"
```

You need to provide your own information for `you` in the file.

12. **Run BitBake With a Target:** Now that you have supplied the `bblayers.conf` file, run the `bitbake` command and provide the target:

```
$ bitbake printhello
Parsing recipes: 100% |#####|
Time: 00:00:00
Parsing of 1 .bb files complete (0 cached, 1 parsed). 1 targets, 0 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
NOTE: Preparing RunQueue
NOTE: Executing RunQueue Tasks

* *
* Hello, World! *
* *

NOTE: Tasks Summary: Attempted 1 tasks of which 0 didn't need to be rerun and all succeeded.
```

BitBake finds the `printhello` recipe and successfully runs the task.

## Note

After the first execution, re-running `bitbake printhello` again will not result in a BitBake run that prints the same console output. The reason for this is that the first time the `printhello.bb` recipe's `do_build` task executes successfully, BitBake writes a stamp file for the task. Thus, the next time you attempt to run the task using that same `bitbake` command, BitBake notices the stamp and therefore determines that the task does not need to be re-run. If you delete the `tmp` directory or run `bitbake -c clean printhello` and then re-run the build, the "Hello, World!" message will be printed again.