Richa

Linux Fou

<richard.purdie@linuxfoundati

Note

Due to production processes, there could be differences between the Yocto Project documentation bundled in the release tarball and the Yocto Project Reference Manual on the Yocto Project website. For the latest version of this manual, see the manual on the website.

Revision History	
Revision 4.0+git	24 November 2010
Released with the Yocto Project 0.9 Release	
Revision 1.0	6 April 2011
Released with the Yocto Project 1.0 Release.	
Revision 1.0.1	23 May 2011
Released with the Yocto Project 1.0.1 Release.	
Revision 1.1	6 October 2011
Released with the Yocto Project 1.1 Release.	
Revision 1.2	April 2012
Released with the Yocto Project 1.2 Release.	
Revision 1.3	October 2012
Released with the Yocto Project 1.3 Release.	
Revision 1.4	April 2013
Released with the Yocto Project 1.4 Release.	
Revision 1.4.1	June 2013
Released with the Yocto Project 1.4.1 Release.	

Revision 1.4.2	August 2013
Released with the Yocto Project 1.4.2 Release.	

Table of Contents

Chapter 1. Introduction

Table of Contents

1.1. Introduction

This manual provides reference information for the current release of the Yocto Project. The Yocto Project is an open-source collaboration project focused on embedded Linux developers. Amongst other things, the Yocto Project uses the OpenEmbedded build system, which is based on the Poky project, to construct complete Linux images. You can find complete introductory and getting started information on the Yocto Project by reading the Yocto Project Quick Start. For task-based information using the Yocto Project, see the Yocto Project Development Manual and the Yocto Project Linux Kernel Development Manual. For Board Support Package (BSP) structure information, see the Yocto Project Board Support Package (BSP) Developer's Guide. You can also find lots of Yocto Project information on the Yocto Project website.

1.2. Documentation Overview

This reference manual consists of the following:

- ***Using the Yocto Project:*** Provides an overview of the components that make up the Yocto Project followed by information about debugging images created in the Yocto Project.
- ***Technical Details:*** Describes fundamental Yocto Project components as well as an explanation behind how the Yocto Project uses shared state (sstate) cache to speed build time.
- ***Directory Structure:*** Describes the Source Directory created either by unpacking a released Yocto Project tarball on your host development system, or by cloning the upstream Poky Git repository.
- ***BitBake:*** Provides an overview of the BitBake tool and its role within the Yocto Project.
- ***Classes:*** Describes the classes used in the Yocto Project.
- ***Images:*** Describes the standard images that the Yocto Project supports.
- ***Features:*** Describes mechanisms for creating distribution, machine, and image features during the build process using the OpenEmbedded build system.
- ***Variables Glossary:*** Presents most variables used by the OpenEmbedded build system, which uses BitBake. Entries describe the function of the variable and how to apply them.
- ***Variable Context:*** Provides variable locality or context.
- ***FAQ:*** Provides answers for commonly asked questions in the Yocto Project development environment.
- ***Contributing to the Yocto Project:*** Provides guidance on how you can contribute back to the Yocto Project.

1.3. System Requirements

For general Yocto Project system requirements, see the "What You Need and How You Get It" section in the Yocto Project Quick Start. The remainder of this section provides details on system requirements not covered in the Yocto Project Quick Start.

1.3.1. Supported Linux Distributions

Currently, the Yocto Project is supported on the following distributions:

- Ubuntu 10.04
- Ubuntu 11.10
- Ubuntu 12.04 (LTS)
- Ubuntu 12.10
- Fedora release 16 (Verne)
- Fedora release 17 (Beefy Miracle)
- Fedora release 18 (Spherical Cow)
- CentOS release 5.6 (Final)
- CentOS release 5.7 (Final)
- CentOS release 5.8 (Final)
- CentOS release 6.3 (Final)
- CentOS release 6.4 (Final)
- Debian GNU/Linux 6.0 (squeeze)
- Debian GNU/Linux 7.0
- openSUSE 11.4
- openSUSE 12.1
- openSUSE 12.2
- openSUSE 12.3

> **Note**
> For additional information on distributions that support the Yocto Project, see the Distribution Support wiki page.

1.3.2. Required Packages for the Host Development System

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.

1.3.2.1. Ubuntu

The following list shows the required packages by function given a supported Ubuntu Linux distribution:

- ***Essentials:*** Packages needed to build an image on a headless system:

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

- ***Graphical Extras:*** Packages recommended if the host system has graphics support:

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

- ***ADT Installer Extras:*** Packages needed if you are going to be using the Application Development Toolkit (ADT) Installer:

```
$ sudo apt-get install autoconf automake libtool libglib2.0-dev
```

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

- ***Graphical Extras:*** Packages recommended if the host system has graphics support:

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

- ***Documentation:*** Packages needed if you are going to build out the Yocto Project documentation manuals:

```
$ sudo yum install make docbook-style-dsssl docbook-style-xsl \
docbook-dtds docbook-utils fop libxslt
```

- ***ADT Installer Extras:*** Packages needed if you are going to be using the Application Development Toolkit (ADT) Installer:

```
$ sudo yum install autoconf automake libtool glib2-devel
```

1.3.2.3. 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 texinfo python-curses patch
```

- ***Graphical Extras:*** Packages recommended if the host system has graphics support:

```
$ sudo zypper install libSDL-devel xterm
```

- ***Documentation:*** Packages needed if you are going to build out the Yocto Project documentation manuals:

```
$ sudo zypper install make fop xsltproc
```

- ***ADT Installer Extras:*** Packages needed if you are going to be using the Application Development Toolkit (ADT) Installer:

```
$ sudo zypper install autoconf automake libtool glib2-devel
```

1.3.2.4. 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 -y install gawk make wget tar bzip2 gzip python unzip perl patch \
diffutils diffstat git cpp gcc gcc-c++ glibc-devel texinfo chrpath
```

- ***Graphical Extras:*** Packages recommended if the host system has graphics support:

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

- ***Documentation:*** Packages needed if you are going to build out the Yocto Project documentation manuals:

```
$ sudo yum -y install make docbook-style-dsssl docbook-style-xsl \
docbook-dtds docbook-utils fop libxslt
```

- ***ADT Installer Extras:*** Packages needed if you are going to be using the Application Development Toolkit (ADT) Installer:

```
$ sudo yum -y install autoconf automake libtool glib2-devel
```

Note

Depending on the CentOS version you are using, other requirements and dependencies might exist. For details, you should look at the CentOS sections on the Poky/GettingStarted/Dependencies wiki page.

1.4. Obtaining the Yocto Project

The Yocto Project development team makes the Yocto Project available through a number of methods:

- ***Releases:*** Stable, tested releases are available through http://downloads.yoctoproject.org/releases/yocto/.
- ***Nightly Builds:*** These releases are available at http://autobuilder.yoctoproject.org/nightly. These builds include Yocto Project releases, meta-toolchain tarball installation scripts, and experimental builds.
- ***Yocto Project Website:*** You can find releases of the Yocto Project and supported BSPs at the Yocto Project website. Along with these downloads, you can find lots of other information at this site.

1.5. Development Checkouts

Development using the Yocto Project requires a local Source Directory. You can set up the Source Directory by downloading a Yocto Project release tarball and unpacking it, or by cloning a copy of the upstream Poky Git repository. For information on both these methods, see the "Getting Set Up" section in the Yocto Project Development Manual.

Chapter 2. Using the Yocto Project

Table of Contents

This chapter describes common usage for the Yocto Project. The information is introductory in nature as other manuals in the Yocto Project documentation set provide more details on how to use the Yocto Project.

2.1. Running a Build

This section provides a summary of the build process and provides information for less obvious aspects of the build process. For general information on how to build an image using the OpenEmbedded build system, see the "Building an Image" section of the Yocto Project Quick Start.

2.1.1. Build Overview

The first thing you need to do is set up the OpenEmbedded build environment by sourcing the environment setup script as follows:

```
$ source oe-init-build-env [<build_dir>]
```

The `build_dir` is optional and specifies the directory the OpenEmbedded build system uses for the build - the Build Directory. If you do not specify a Build Directory, it defaults to `build` in your current working directory. 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. See the "`oe-init-build-env`" section for more information on this script.

Once the build environment is set up, you can build a target using:

```
$ bitbake <target>
```

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`, etc. 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.

> **Note**
> Building an image without GNU General Public License Version 3 (GPLv3) components is only supported for minimal and base images. See the "Images" chapter for more information.

2.1.2. Building an Image Using GPL Components

When building an image using GPL components, you need to maintain your original settings and not switch back and forth applying different versions of the GNU General Public License. If you rebuild using different versions of GPL, dependency errors might occur due to some components not being rebuilt.

2.2. Installing and Using the Result

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 "Using Pre-Built Binaries and QEMU" section in the Yocto Project Quick Start. For information about how to install these images, see the documentation for your particular board or machine.

2.3. Debugging Build Failures

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.

For discussions on debugging, see the "Debugging With the GNU Project Debugger (GDB) Remotely" and "Working within Eclipse" sections in the Yocto Project Development Manual.

2.3.1. Task Failures

The log file for shell tasks is available in `${WORKDIR}/temp/log.do_taskname.pid`. For example, the `compile` task for the QEMU minimal image for the x86 machine (`qemux86`) might be `tmp/work/qemux86-poky-linux/core-image-minimal/1.0-r0/temp/log.do_compile.20830`. To see what BitBake runs to generate that log, look at the corresponding `run.do_taskname.pid` file located in the same directory.

Presently, the output from Python tasks is sent directly to the console.

2.3.2. Running Specific Tasks

Any given package consists of a set of tasks. The standard BitBake behavior in most cases is: `fetch`, `unpack`, `patch`, `configure`, `compile`, `install`, `package`, `package_write`, and `build`. The default task is `build` and any tasks on which it depends build first. Some tasks exist, such as `devshell`, that 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
```

If you wish to rerun a task, use the `-f` force option. For example, the following sequence forces recompilation after changing files in the working directory.

```
$ bitbake matchbox-desktop
          .
          .
   [make some changes to the source code in the working 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 `compile` task was rerun and therefore understands that the other tasks also need to be run again.

You can view a list of tasks in a given package by running the `listtasks` task as follows:

```
$ bitbake matchbox-desktop -c listtasks
```

The results are in the file `${WORKDIR}/temp/log.do_listtasks`.

2.3.3. Dependency Graphs

Sometimes it can be hard to see why BitBake wants to build some other packages before a given package you have specified. The `bitbake -g targetname` command creates the `depends.dot`, `package-depends.dot`, and `task-depends.dot` files in the current directory. These files show the package and task dependencies and are useful for debugging problems. You can use the `bitbake -g -u depexp targetname` command to display the results in a more human-readable form.

2.3.4. 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.

2.3.5. Development Host System Issues

Sometimes issues on the host development system can cause your build to fail. Following are known, host-specific problems. Be sure to always consult the Release Notes for a look at all release-related issues.

- ***`eglibc-initial`* fails to build**: If your development host system has the unpatched `GNU Make 3.82`, the `do_install` task fails for `eglibc-initial` during the build.

  Typically, every distribution that ships `GNU Make 3.82` as the default already has the patched version. However, some distributions, such as Debian, have `GNU Make 3.82` as an option, which is unpatched. You will see this error on these types of distributions. Switch to `GNU Make 3.81` or patch your `make` to solve the problem.

2.3.6. Building with No Dependencies

If you really want to build a specific `.bb` file, you can use the command form `bitbake -b <somepath/somefile.bb>`. This command form does not check for dependencies so you should use it only when you know its dependencies already exist. You can also specify fragments of the filename. In this case, BitBake checks for a unique match.

2.3.7. Variables

You can use the `-e` BitBake option to display the resulting environment for a configuration when you do not specify a package or for a specific package when you do specify the package. If you want to show the environment resulting from parsing a single recipe, use the `-b recipename` form.

2.3.8. Recipe Logging Mechanisms

Best practices exist while writing recipes that both log build progress and act on build conditions such as warnings and errors. Both Python and Bash language bindings exist for the logging mechanism:

- ***Python:*** For Python functions, BitBake supports several loglevels: `bb.fatal`, `bb.error`, `bb.warn`, `bb.note`, `bb.plain`, and `bb.debug`.
- ***Bash:*** For Bash functions, the same set of loglevels exist and are accessed with a similar syntax: `bbfatal`, `bberror`, `bbwarn`, `bbnote`, `bbplain`, and `bbdebug`.

For guidance on how logging is handled in both Python and Bash recipes, see the `logging.bbclass` file in the `meta/classes` folder of the Source Directory.

2.3.8.1. 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:

```
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")
}
```

2.3.8.2. 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"
}
```

2.3.9. Other 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).

2.4. 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

2.4.1. Enabling and Disabling Build History

Build history is disabled by default. To enable it, add the following statements to 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 build process to collect build output information and commit it 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.

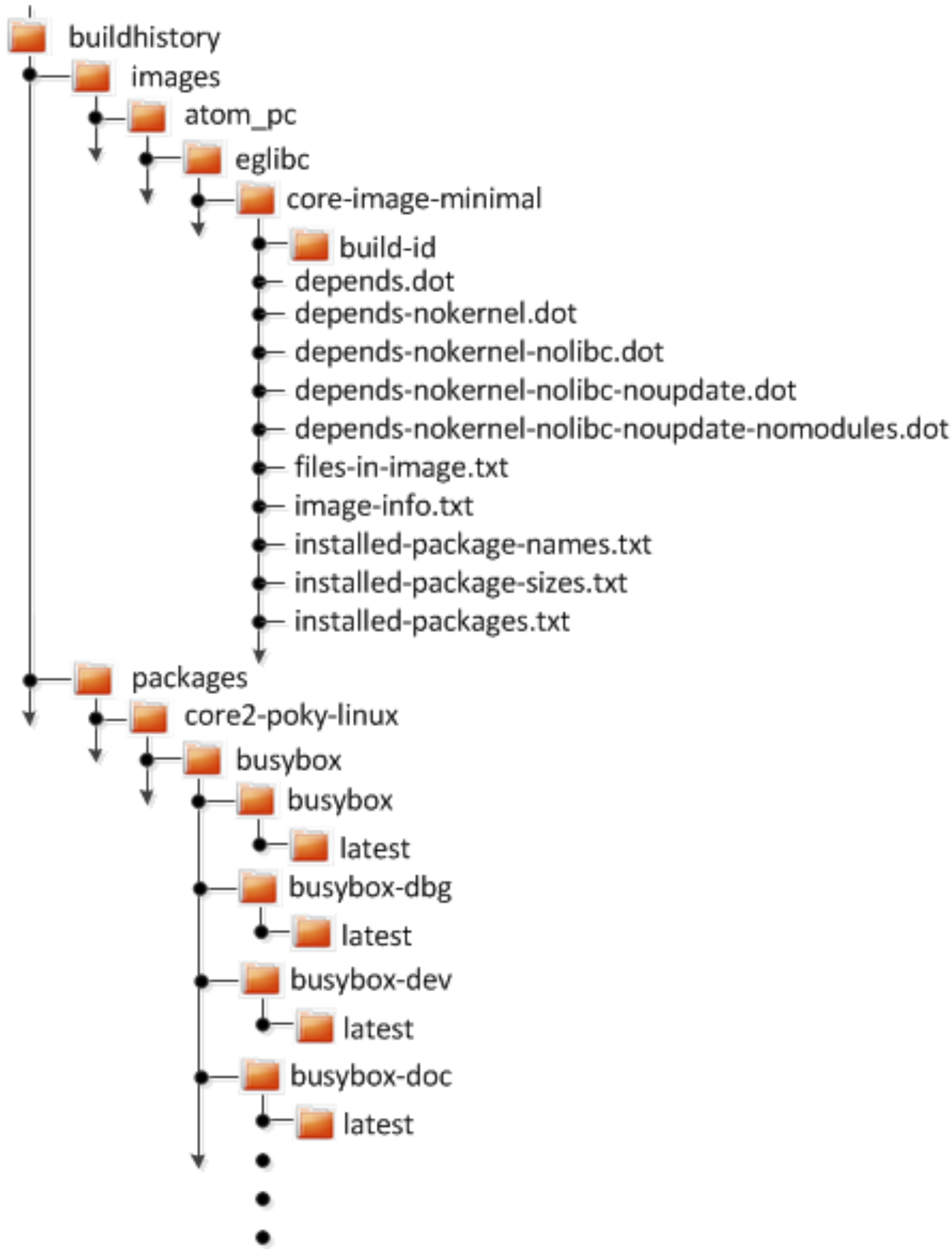
You can disable build history by removing the previous statements from your `conf/local.conf` file. However, you should realize that enabling and disabling build history in this manner can change the `do_package` task checksums, which if you are using the OEBasicHash signature generator (the default for many current distro configurations including `DISTRO = "poky"` and `DISTRO = ""`) and will result in the packaging tasks being re-run during the subsequent build.

To disable the build history functionality without causing the packaging tasks to be re-run, add this statement to your `conf/local.conf` file:

```
BUILDHISTORY_FEATURES = ""
```

2.4.2. Understanding What the Build History Contains

Build history information is kept in `$TMPDIR/buildhistory` in the Build Directory. The following is an example abbreviated listing:



At the top level, there is a `metadata-revs` file that lists the revisions of the repositories for the layers enabled when the build was produced. The rest of the data splits into separate `packages`, `images` and `sdk` directories, the contents of which are described below.

2.4.2.1. 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/core2-poky-linux/busybox/busybox/latest` contains the following:

```
PV = 1.19.3
PR = r3
RDEPENDS = update-rc.d eglibc (>= 2.13)
RRECOMMENDS = busybox-syslog busybox-udhcpc
PKGSIZE = 564701
FILES = /usr/bin/* /usr/sbin/* /usr/libexec/* /usr/lib/lib*.so.* \
   /etc /com /var /bin/* /sbin/* /lib/*.so.* /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 = /etc/busybox.links /etc/init.d/hwclock.sh /bin/busybox /bin/sh
```

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.

There is also a file corresponding to the recipe from which the package came (e.g. buildhistory/packages/core2-poky-linux/busybox/latest):

```
PV = 1.19.3
PR = r3
DEPENDS = virtual/i586-poky-linux-gcc virtual/i586-poky-linux-compilerlibs \
   virtual/libc update-rc.d-native
PACKAGES = busybox-httpd busybox-udhcpd busybox-udhcpc busybox-syslog \
   busybox-mdev busybox-dbg busybox busybox-doc busybox-dev \
   busybox-staticdev busybox-locale
```

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/emenlow-poky-linux/linux-yocto/latest_srcrev):

```
# SRCREV_machine = "b5c37fe6e24eec194bb29d22fdd55d73bcc709bf"
SRCREV_machine = "b5c37fe6e24eec194bb29d22fdd55d73bcc709bf"
# SRCREV_emgd = "caea08c988e0f41103bbe18eafca20348f95da02"
SRCREV_emgd = "caea08c988e0f41103bbe18eafca20348f95da02"
# SRCREV_meta = "c2ed0f16fdec628242a682897d5d86df4547cf24"
SRCREV_meta = "c2ed0f16fdec628242a682897d5d86df4547cf24"
```

You can use the buildhistory-collect-srcrevs command 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:

```
# emenlow-poky-linux
SRCREV_machine_pn-linux-yocto = "b5c37fe6e24eec194bb29d22fdd55d73bcc709bf"
SRCREV_emgd_pn-linux-yocto = "caea08c988e0f41103bbe18eafca20348f95da02"
SRCREV_meta_pn-linux-yocto = "c2ed0f16fdec628242a682897d5d86df4547cf24"
# core2-poky-linux
SRCREV_pn-kmod = "62081c0f68905b22f375156d4532fd37fa5c8d33"
SRCREV_pn-blktrace = "d6918c8832793b4205ed3bfede78c2f915c23385"
SRCREV_pn-opkg = "649"
```

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 was 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 shown above (e.g., emenlow-poky-linux).

2.4.2.2. Build History Image Information

The files produced for each image are as follows:

- `build-id`: Human-readable information about the build configuration and metadata source revisions.
- `*.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 = 1.1+snapshot-20120207
USER_CLASSES = image-mklibs image-prelink
IMAGE_CLASSES = image_types
IMAGE_FEATURES = debug-tweaks x11-base apps-x11-core \
   package-management ssh-server-dropbear package-management
IMAGE_LINGUAS = en-us en-gb
IMAGE_INSTALL = task-core-boot task-base-extended
BAD_RECOMMENDATIONS =
ROOTFS_POSTPROCESS_COMMAND = buildhistory_get_image_installed ;   rootfs_update_timestamp ;
IMAGE_POSTPROCESS_COMMAND = buildhistory_get_imageinfo ;
IMAGESIZE = 171816
```

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.

2.4.2.3. 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 history or any package 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"
```

2.4.2.4. Build History SDK Information

Build history collects similar information on the contents of SDKs (e.g., `meta-toolchain` or `bitbake -c populate_sdk imagename`) as compared to information it collects for images. The following list shows the files produced for each SDK:

- `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.
- 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:

- `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-eglibc-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.

2.4.2.5. 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
```

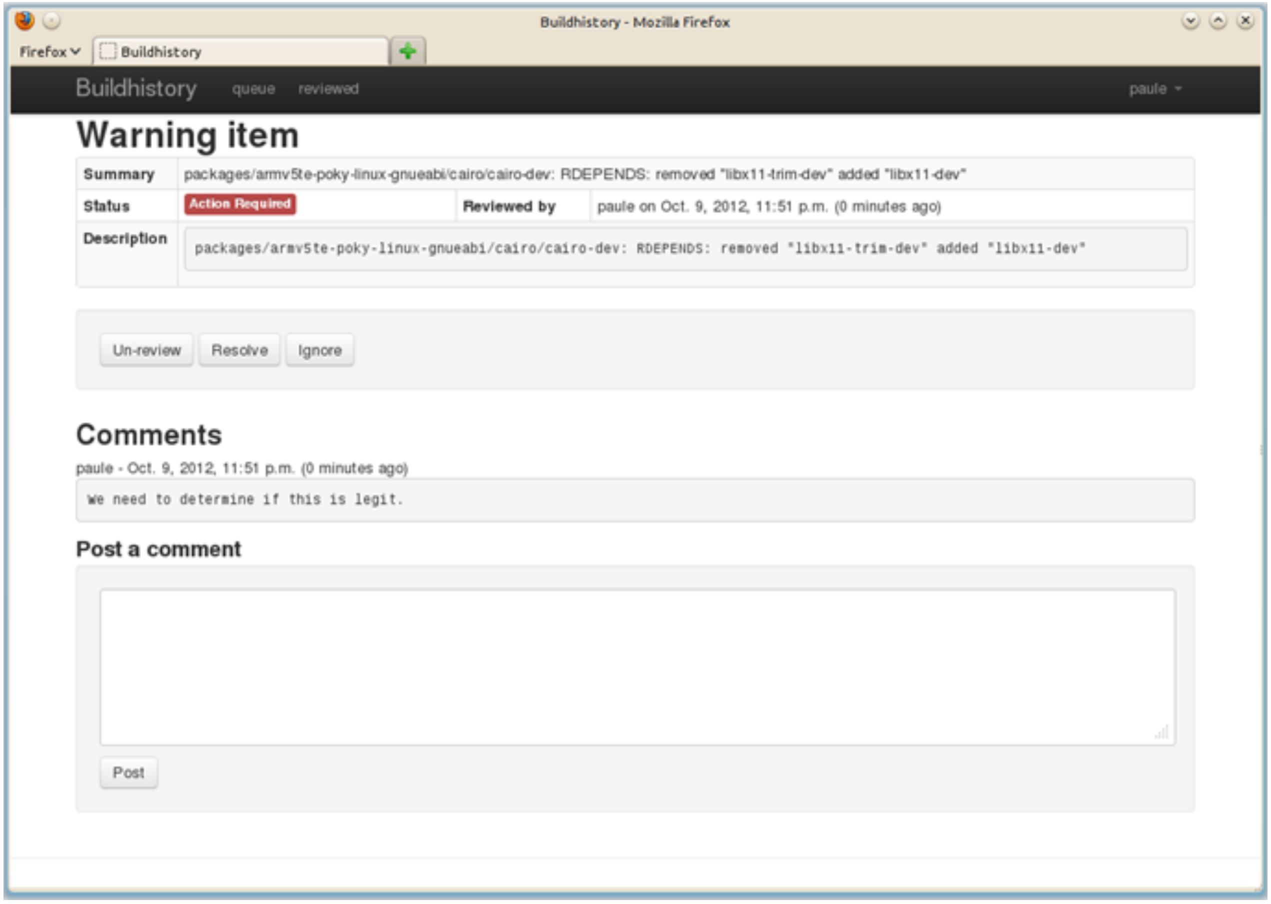
You need to realize, however, that this method does show changes that are not significant (e.g. a package's size changing by a few bytes).

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/eglibc/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/eglibc/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"
```

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:



Chapter 3. Technical Details

Table of Contents

This chapter provides technical details for various parts of the Yocto Project. Currently, topics include Yocto Project components, shared state (sstate) cache, x32, and Licenses.

3.1. Yocto Project Components

The BitBake task executor together with various types of configuration files form the OpenEmbedded Core. This section overviews these by describing what they are used for 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.

For more information on data, see the "Yocto Project Terms" section in the Yocto Project Development Manual.

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 Manual.

Following are some brief details on these core components. For more detailed information on these components, see the "Source Directory Structure" chapter.

3.1.1. 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. To see a list of the options BitBake supports, use the following help command:

```
$ 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" in this manual). The target often equates to the first part of a `.bb` filename. So, to run the `matchbox-desktop_1.2.3.bb` 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 and Providers" section.

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

> **Note**
> This release of the Yocto Project does not support the `glibc` GNU version of the Unix standard C library. By default, the OpenEmbedded build system builds with `eglibc`.

A useful BitBake option to consider is the `-k` or `--continue` option. This option instructs BitBake to try and continue processing the job as much 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.

3.1.2. Metadata (Recipes)

The `.bb` files are usually referred to as "recipes." In general, a recipe contains information about a single piece of software. The information includes the location from which to download the source patches (if any are needed), which special configuration options to apply, how to compile the source files, and how to package the compiled output.

The term "package" can also be used to describe recipes. However, since the same word 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.

3.1.3. Classes

Class files (`.bbclass`) contain information that is useful to share between Metadata files. An example is the Autotools class, which contains common settings for any application that Autotools uses. The "Classes" chapter provides details about common classes and how to use them.

3.1.4. Configuration

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 `local.conf`, which is found in the Build Directory.

3.2. 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 there is no possibility of stale data causing problems. When developers hit problems, they typically default back to building from scratch so they know the state of things 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 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.

Note

The OpenEmbedded 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 OpenEmbedded works with packages and can track incrementing `PR` information, see the "Incrementing a Package Revision Number" section.

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

3.2.1. Overall Architecture

When determining what parts of the system need to be built, BitBake uses a per-task basis and does not use 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, `do_install` and `do_package` output 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 situation. 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.

3.2.2. 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 working 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. The checksum therefore needs to exclude WORKDIR. The simplistic approach for excluding the working 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 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 WORKDIR 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. 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, 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 statements effectively result in a list of global variable dependency excludes - variables never included in any checksum:

```
BB_HASHBASE_WHITELIST ?= "TMPDIR FILE PATH PWD BB_TASKHASH BBPATH"
BB_HASHBASE_WHITELIST += "DL_DIR SSTATE_DIR THISDIR FILESEXTRAPATHS"
BB_HASHBASE_WHITELIST += "FILE_DIRNAME HOME LOGNAME SHELL TERM USER"
BB_HASHBASE_WHITELIST += "FILESPATH USERNAME STAGING_DIR_HOST STAGING_DIR_TARGET"
```

The previous example actually excludes WORKDIR since it 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, 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
```

3.2.3. Shared State

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

The shared state class (`sstate.bbclass`) 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 - the build process does not need to worry about its source.

There are two types of output, one 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.bbclass`. From a user's perspective, adding shared state wrapping to a task is as simple as this `do_deploy` example taken from `do_deploy.bbclass`:

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

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

In the example, we add some extra flags to the task, a name field ("deploy"), an input directory where the task sends data, and the output directory where the data from the task should eventually be copied. We also add a `_setscene` variant of the task and add the task name to the `SSTATETASKS` list.

If you have a directory whose contents you need to preserve, you can do this with a line like the following:

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

This method, as well as the following example, also works for multiple directories.

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

These methods also include the ability to take a lockfile when manipulating shared state directory structures since some cases are sensitive to file additions or removals.

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 \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.

3.2.4. Tips and Tricks

The code in the build system that supports incremental builds is not simple code. This section presents some tips and tricks that help you work around issues related to shared state code.

3.2.4.1. Debugging

When things go wrong, debugging needs to be straightforward. Because of this, the Yocto Project team included strong debugging tools:

- Whenever a shared state package is written out, so is a corresponding `.siginfo` file. This practice results in a pickled Python database of all the metadata that went into creating the hash for a given shared state package.
- If you run BitBake with the `--dump-signatures` (or `-S`) option, BitBake dumps out `.siginfo` files in the stamp directory for every task it would have executed instead of building the specified target package.
- There is a `bitbake-diffsigs` command that can process `.siginfo` files. If you specify one of these files, BitBake dumps out the dependency information in the file. If you specify two files, BitBake compares the two files and dumps out the differences between the two. This more easily helps answer the question of "What changed between X and Y?"

3.2.4.2. Invalidating Shared State

The shared state code uses checksums and shared state cache to avoid unnecessarily rebuilding tasks. As with all schemes, this one has some drawbacks. It is possible that you could make implicit changes that are not factored into the checksum calculation, but do affect a task's output. A good example is perhaps when a tool changes its output. Assume that the output of `rpmdeps` needed to change. The result of the change should be that all the `package`, `package_write_rpm`, and `package_deploy-rpm` shared state cache items would become invalid. But, because this is a change that 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 change you make. Note that any changes you make directly to a function automatically are factored into the checksum calculation and thus, will invalidate the associated area of sstate cache. 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. Once you are aware of such changes, you can take steps to invalidate the cache and force the tasks to run. The steps to take are as simple as changing 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. The change is purely cosmetic, but it causes the checksum to be recalculated and forces the task to be run again.

Note

For an example of a commit that makes a cosmetic change to invalidate a shared state, see this commit.

3.3. x32

x32 is a processor-specific Application Binary Interface (psABI) for x86_64. 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.

3.3.1. Support

While the x32 psABI specifications are not fully finalized, this Yocto Project release supports current development specifications of x32 psABI. As of this release of the Yocto Project, x32 psABI support exists as follows:

- You can create packages and images in x32 psABI format on x86_64 architecture targets.
- You can successfully build many recipes with the x32 toolchain.
- You can create and boot `core-image-minimal` and `core-image-sato` images.

3.3.2. Stabilizing and Completing x32

As of this Yocto Project release, the x32 psABI kernel and library interfaces specifications are not finalized.

Future Plans for the x32 psABI in the Yocto Project include the following:

- Enhance and fix the few remaining recipes so they work with and support x32 toolchains.
- Enhance RPM Package Manager (RPM) support for x32 binaries.
- Support larger images.

3.3.3. Using x32 Right Now

Follow these steps to use the x32 spABI:

- Enable the x32 psABI tuning file for `x86_64` machines by editing the `conf/local.conf` like this:

```
MACHINE = "qemux86-64"
```

```
DEFAULTTUNE = "x86-64-x32"
baselib = "${@d.getVar('BASE_LIB_tune-' + (d.getVar('DEFAULTTUNE', True) \
   or 'INVALID'), True) or 'lib'}"
#MACHINE = "atom-pc"
#DEFAULTTUNE = "core2-64-x32"
```

- As usual, use BitBake to build an image that supports the x32 psABI. Here is an example:

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

- As usual, run your image using QEMU:

  ```
  $ runqemu qemux86-64 core-image-sato
  ```

3.4. Wayland

Wayland is a computer display server protocol that when implemented 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 it release. This section describes what you need to do to implement Wayland and use the compositor when building an image for a supporting target.

3.4.1. Support

The Wayland protocol libraries and the reference Weston compositor ship as 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.

3.4.2. Enabling Wayland in an Image

To enable Wayland, you need to enable it to be built and enable it to be included in the image.

3.4.2.1. 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

3.4.2.2. 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"
```

3.4.3. 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 interpretor (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
   ```

3.5. Licenses

This section describes the mechanism by which the OpenEmbedded build system tracks changes to licensing text. The section also describes how to enable commercially licensed recipes, which by default are disabled.

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

3.5.1. 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.

3.5.1.1. 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 \
                    ..."
```

The build system uses the S variable as the default directory used when searching files listed in LIC_FILES_CHKSUM. The previous example employs the default directory.

You can also use relative paths as shown in the following example:

```
LIC_FILES_CHKSUM = "file://src/ls.c;beginline=5;endline=16;\
                                    md5=bb14ed3c4cda583abc85401304b5cd4e"
LIC_FILES_CHKSUM = "file://../license.html;md5=5c94767cedb5d6987c902ac850ded2c6"
```

In this example, the first line locates a file in ${S}/src/ls.c. The second line refers to a file in WORKDIR, which is the parent of S.

Note that LIC_FILES_CHKSUM variable is mandatory for all recipes, unless the LICENSE variable is set to "CLOSED".

3.5.1.2. 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.

> **Tip**
> 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.

> **Tip**
> If the whole file contains only license text, you do not need to use the "beginline" and "endline" parameters.

3.5.2. 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 $HOME/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 $HOME/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"
```

3.5.2.1. 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_FLAG 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_FLAG 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.

3.5.2.2. Other Variables Related to Commercial Licenses

Other helpful variables related to commercial license handling exist and are defined in the `$HOME/poky/meta/conf/distro/include/default-distrovars.inc` file:

```
COMMERCIAL_AUDIO_PLUGINS ?= ""
COMMERCIAL_VIDEO_PLUGINS ?= ""
COMMERCIAL_QT = ""
```

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"
COMMERCIAL_QT ?= "qmmp"
LICENSE_FLAGS_WHITELIST = "commercial_gst-plugins-ugly commercial_gst-plugins-bad commercial_qm
```

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 plug-ins as part of the `COMMERCIAL_AUDIO_PLUGINS` and `COMMERCIAL_VIDEO_PLUGINS` statements or commercial Qt components as part of the `COMMERCIAL_QT` statement (along with the enabling `LICENSE_FLAGS_WHITELIST`) includes the plug-ins or components into built images, thus adding support for media formats or components.

Chapter 4. Migrating to a Newer Yocto Project Release

Table of Contents

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.

4.1. 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.

4.1.1. Local Configuration

Differences include changes for SSTATE_MIRRORS and bblayers.conf.

4.1.1.1. SSTATE_MIRRORS

The shared state cache (sstate-cache), as pointed to by SSTATE_DIR, by default now has two-character subdirectories to prevent issues rising from too many files in the same directory. Also, native sstate-cache packages 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"
```

4.1.1.2. 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 or Hob 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.

4.1.2. Recipes

Differences include changes for the following:

- Python function whitespace
- proto= in SRC_URI
- nativesdk
- Task recipes
- IMAGE_FEATURES
- Removed recipes

4.1.2.1. 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.

4.1.2.2. 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.

4.1.2.3. 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 and any references need to be updated to use `nativesdk-*` instead of `*-nativesdk`.

4.1.2.4. 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 "Package Groups - packagegroup.bbclass" section for further details.

4.1.2.5. 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.

4.1.2.6. 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.

4.2. 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.

4.2.1. 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.

4.2.2. Build Behavior

Differences include the following:

- ***Shared State Code:*** The shared state code has been optimized to avoid running unnecessary tasks. For example, `bitbake -c rootfs some-image` from shared state no longer populates the target sysroot since that is not necessary. 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.

4.2.3. 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.

4.2.4. 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" in the Yocto Project Development Manual.

4.2.5. 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.

4.2.6. 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.

4.2.7. 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
```

4.2.8. 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.

4.2.9. 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 / xfontcacheproto*:** 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.

Chapter 5. Source Directory Structure

Table of Contents

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 "Getting Set Up" section in the Yocto Project Development 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.

5.1. Top-Level Core Components

This section describes the top-level components of the Source Directory.

5.1.1. `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 wrapper script in `scripts/` is executed to run the main BitBake executable, which resides in the `bitbake/bin/` directory. Sourcing the `oe-init-build-env` script 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 documentation included in the `bitbake/doc/manual` directory of the Source Directory.

5.1.2. `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 `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.

5.1.3. `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 `ref-manual`.

5.1.4. `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.)

5.1.5. `meta-yocto/`

This directory contains the configuration for the Poky reference distribution.

5.1.6. `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.

5.1.7. `meta-hob/`

This directory contains template recipes used by Hob, which is a Yocto Project build user interface. For more information on the Hob, see the Hob Project webpage.

5.1.8. `meta-skeleton/`

This directory contains template recipes for BSP and kernel development.

5.1.9. `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 contributing back to the Yocto Project, such as `create_pull_request` and `send_pull_request`.

5.1.10. `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 this script before running BitBake commands. The script uses other scripts within the `scripts` directory to do the bulk of the work.

By default, running this script without a Build Directory argument creates the `build` directory. If you provide a Build Directory argument when you `source` the script, you direct 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
```

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.

5.1.11. LICENSE, README, and README.hardware

These files are standard top-level files.

5.2. The Build Directory - build/

The OpenEmbedded build system creates the Build Directory during the build. By default, this directory is named `build`.

5.2.1. build/pseudodone

This tag file indicates that the initial pseudo binary was created. The file is built the first time BitBake is invoked.

5.2.2. build/conf/local.conf

This file contains all the local user configuration for your build environment. If there is no `local.conf` present, it is created from `local.conf.sample`. 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`), the location from which you want to downloaded files (`DL_DIR`), and how you want your host machine to use resources (`BB_NUMBER_THREADS` and `PARALLEL_MAKE`).

5.2.3. build/conf/bblayers.conf

This file defines layers, which are directory trees, traversed (or walked) by BitBake. If `bblayers.conf` is not present, it is created from `bblayers.conf.sample` when you `source` the environment setup script.

The `bblayers.conf` file uses the `BBLAYERS` variable to list the layers BitBake tries to find. The file uses the `BBLAYERS_NON_REMOVABLE` variable to list layers that must not be removed.

5.2.4. build/conf/sanity_info

This file indicates the state of the sanity checks and is created during the build.

5.2.5. 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.

5.2.6. `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.

5.2.7. `build/tmp/`

This directory receives all the OpenEmbedded build system's output. 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.

5.2.8. `build/tmp/buildstats/`

This directory stores the build statistics.

5.2.9. `build/tmp/cache/`

When BitBake parses the metadata, it creates a cache file of the result that can be used when subsequently running commands. BitBake stores these results here on a per-machine basis.

5.2.10. `build/tmp/deploy/`

This directory contains any "end result" output from the OpenEmbedded build process.

5.2.11. `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.

5.2.12. `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.

5.2.13. `build/tmp/deploy/licenses/`

This directory receives package licensing information. For example, the directory contains sub-directories for `bash`, `busybox`, and `eglibc` (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.

5.2.14. `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-*`, `hob-image-*`, etc.). 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
```

5.2.15. build/tmp/deploy/ipk/

This directory receives .ipk packages produced by the build process.

5.2.16. build/tmp/sysroots/

This directory contains shared header files and libraries as well as other shared data. Packages that need to share output with other packages do so within this directory. The directory is subdivided by architecture so multiple builds can run within the one Build Directory.

5.2.17. 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.

5.2.18. 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 check_pkg or distro_check tasks. Running a build does not necessarily mean this directory is created.

5.2.19. build/tmp/pkgdata/

This directory contains intermediate packaging data that is used later in the packaging process. For more information, see the "Packaging - package*.bbclass" section.

5.2.20. 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 a Quilt Flow" section in the Yocto Project Development 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/.

5.3. The Metadata - meta/

As mentioned previously, Metadata is the core of the Yocto Project. Metadata has several important subdivisions:

5.3.1. 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.

5.3.2. `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.

5.3.3. `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.

5.3.4. `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.

5.3.5. `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 knows permissions.

5.3.6. `meta/lib/`

This directory contains OpenEmbedded Python library code used during the build process.

5.3.7. `meta/recipes-bsp/`

This directory contains anything linking to specific hardware or hardware configuration information such as "u-boot" and "grub".

5.3.8. `meta/recipes-connectivity/`

This directory contains libraries and applications related to communication with other devices.

5.3.9. `meta/recipes-core/`

This directory contains what is needed to build a basic working Linux image including commonly used dependencies.

5.3.10. `meta/recipes-devtools/`

This directory contains tools that are primarily used by the build system. The tools, however, can also be used on targets.

5.3.11. `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.

5.3.12. `meta/recipes-gnome/`

This directory contains all things related to the GTK+ application framework.

5.3.13. `meta/recipes-graphics/`

This directory contains X and other graphically related system libraries

5.3.14. `meta/recipes-kernel/`

This directory contains the kernel and generic applications and libraries that have strong kernel dependencies.

5.3.15. `meta/recipes-lsb4/`

This directory contains recipes specifically added to support the Linux Standard Base (LSB) version 4.x.

5.3.16. `meta/recipes-multimedia/`

This directory contains codecs and support utilities for audio, images and video.

5.3.17. `meta/recipes-qt/`

This directory contains all things related to the Qt application framework.

5.3.18. `meta/recipes-rt/`

This directory contains package and image recipes for using and testing the `PREEMPT_RT` kernel.

5.3.19. `meta/recipes-sato/`

This directory contains the Sato demo/reference UI/UX and its associated applications and configuration data.

5.3.20. `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).

5.3.21. `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.

5.3.22. `meta/recipes.txt`

This file is a description of the contents of `recipes-*`.

Chapter 6. BitBake

Table of Contents

BitBake is a program written in Python that interprets the Metadata used by the OpenEmbedded build system. At some point, developers wonder what actually happens when you enter:

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

This chapter provides an overview of what happens behind the scenes from BitBake's perspective.

Note

BitBake strives to be a generic "task" executor that is capable of handling complex dependency relationships. As such, it has no real knowledge of what the tasks being executed actually do. BitBake just considers a list of tasks with dependencies and handles metadata that consists of variables in a certain format that get passed to the tasks.

6.1. Parsing

BitBake parses configuration files, classes, and `.bb` files.

The first thing BitBake does is look for the `bitbake.conf` file. This file resides in the Source Directory within the `meta/conf/` directory. BitBake finds it by examining its `BBPATH` environment variable and looking for the `meta/conf/` directory.

The `bitbake.conf` file lists other configuration files to include from a `conf/` directory below the directories listed in `BBPATH`. In general, the most important configuration file from a user's perspective is `local.conf`, which contains a user's customized settings for the OpenEmbedded build environment. Other notable configuration files are the distribution configuration file (set by the `DISTRO` variable) and the machine configuration file (set by the `MACHINE` variable). The `DISTRO` and `MACHINE` BitBake environment variables are both usually set in the `local.conf` file. Valid distribution configuration files are available in the `meta/conf/distro/` directory and valid machine configuration files in the `meta/conf/machine/` directory. Within the `meta/conf/machine/include/` directory are various `tune-*.inc` configuration files that provide common "tuning" settings specific to and shared between particular architectures and machines.

After the parsing of the configuration files, some standard classes are included. The `base.bbclass` file is always included. Other classes that are specified in the configuration using the `INHERIT` variable are also included. Class files are searched for in a `classes` subdirectory under the paths in `BBPATH` in the same way as configuration files.

After classes are included, the variable `BBFILES` is set, usually in `local.conf`, and defines the list of places to search for `.bb` files. By default, the `BBFILES` variable specifies the `meta/recipes-*/` directory within Poky. Adding extra content to `BBFILES` is best achieved through the use of BitBake layers as described in the "Understanding and Creating Layers" section of the Yocto Project Development Manual.

BitBake parses each `.bb` file in `BBFILES` and stores the values of various variables. In summary, for each `.bb` file the configuration plus the base class of variables are set, followed by the data in the `.bb` file itself, followed by any inherit commands that `.bb` file might contain.

Because parsing `.bb` files is a time consuming process, a cache is kept to speed up subsequent parsing. This cache is invalid if the timestamp of the `.bb` file itself changes, or if the timestamps of any of the include, configuration files or class files on which the `.bb` file depends change.

6.2. Preferences and Providers

Once all the `.bb` files have been parsed, BitBake starts to build the target (`core-image-sato` in the previous section's example) and looks for providers of that target. Once a provider is selected, BitBake resolves all the dependencies for the target. In the case of `core-image-sato`, it would lead to `packagegroup-core-x11-sato`, which in turn leads to recipes like `matchbox-terminal`, `pcmanfm` and `gthumb`. These recipes in turn depend on `eglibc` and the toolchain.

Sometimes a target might have multiple providers. A common example is "virtual/kernel", which is provided by each kernel package. 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.

Understanding how providers are chosen is made complicated by the fact that multiple versions might exist. 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 (usually in the distro configuration). You can influence the order by using the `DEFAULT_PREFERENCE` variable. By default, files have a preference of "0". Setting the `DEFAULT_PREFERENCE` to "-1" makes the package unlikely to be used unless it is explicitly referenced. Setting the `DEFAULT_PREFERENCE` to "1" makes it likely the package is used. `PREFERRED_VERSION` overrides any `DEFAULT_PREFERENCE` setting. `DEFAULT_PREFERENCE` is often used to mark newer and more experimental package versions until they have undergone sufficient testing to be considered stable.

In summary, BitBake has created a list of providers, which is prioritized, for each target.

6.3. 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 BitBake documentation, which is found in the `bitbake/doc/manual` directory within the Source Directory. At a basic level, it is sufficient to know that BitBake uses the `DEPENDS` and `RDEPENDS` variables when calculating dependencies.

6.4. 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 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. See the "Building an Image" section in the Yocto Project Quick Start for more information.

As each task completes, a timestamp is written to the directory specified by the `STAMP` variable. On subsequent runs, BitBake looks within the `/build/tmp/stamps` directory 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 `.bb` 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. This behavior could change or become configurable in future versions of BitBake.

Note

Some tasks are marked as "nostamp" tasks. No timestamp file is created when these tasks are run. Consequently, "nostamp" tasks are always rerun.

6.5. Running a Task

Tasks can either be a shell task or a Python task. For shell tasks, BitBake writes a shell script to `${WORKDIR}/temp/run.do_taskname.pid` and then executes the script. The generated shell script contains all the exported variables, and the shell functions with all variables expanded. Output from the shell script goes to the file `${WORKDIR}/temp/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.

Once all the tasks have been completed BitBake exits.

When running a task, BitBake tightly controls the execution environment of the build tasks to make sure unwanted contamination from the build machine cannot influence the build. Consequently, if you do want something to get passed into the build task's environment, you must take a few steps:

1. Tell BitBake to load what you want from the environment into the data store. You can do so through the `BB_ENV_EXTRAWHITE` variable. For example, assume you want to prevent the build system from accessing your `$HOME/.ccache` directory. The following command tells BitBake to load `CCACHE_DIR` from the environment into the data store:

   ```
   export BB_ENV_EXTRAWHITE="$BB_ENV_EXTRAWHITE CCACHE_DIR"
   ```

2. Tell BitBake to export what you have loaded into the environment store to the task environment of every running task. Loading something from the environment into the data store (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.conf` or distro 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 shared state checksums. If doing so results in unnecessary rebuilds of tasks, you can whitelist the variable so that the shared state code ignores the dependency when it creates checksums. For information on this process, see the `BB_HASHBASE_WHITELIST` example in the "Checksums (Signatures)" section.

6.6. BitBake Command Line

Following is the BitBake help output:

```
$ bitbake --help
Usage: bitbake [options] [package ...]

Executes the specified task (default is 'build') for a given set of BitBake files.
It expects that BBFILES is defined, which is a space separated list of files to
be executed.  BBFILES does support wildcards.
Default BBFILES are the .bb files in the current directory.

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -b BUILDFILE, --buildfile=BUILDFILE
                        execute the task against this .bb file, rather than a
                        package from BBFILES. Does not handle any
```

```
                        dependencies.
  -k, --continue        continue as much as possible after an error. While the
                        target that failed, and those that depend on it,
                        cannot be remade, the other dependencies of these
                        targets can be processed all the same.
  -a, --tryaltconfigs   continue with builds by trying to use alternative
                        providers where possible.
  -f, --force           force run of specified cmd, regardless of stamp status
  -c CMD, --cmd=CMD     Specify task to execute. Note that this only executes
                        the specified task for the providee and the packages
                        it depends on, i.e. 'compile' does not implicitly call
                        stage for the dependencies (IOW: use only if you know
                        what you are doing). Depending on the base.bbclass a
                        listtasks tasks is defined and will show available
                        tasks
  -r PREFILE, --read=PREFILE
                        read the specified file before bitbake.conf
  -R POSTFILE, --postread=POSTFILE
                        read the specified file after bitbake.conf
  -v, --verbose         output more chit-chat to the terminal
  -D, --debug           Increase the debug level. You can specify this more
                        than once.
  -n, --dry-run         don't execute, just go through the motions
  -S, --dump-signatures
                        don't execute, just dump out the signature
                        construction information
  -p, --parse-only      quit after parsing the BB files (developers only)
  -s, --show-versions   show current and preferred versions of all packages
  -e, --environment     show the global or per-package environment (this is
                        what used to be bbread)
  -g, --graphviz        emit the dependency trees of the specified packages 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 print a report
  -u UI, --ui=UI        userinterface to use
  -t SERVERTYPE, --servertype=SERVERTYPE
                        Choose which server to use, none, process or xmlrpc
  --revisions-changed   Set the exit code depending on whether upstream
                        floating revisions have changed or not
```

6.7. Fetchers

BitBake also contains a set of "fetcher" modules that allow retrieval of source code from various types of sources. For example, BitBake can get source code from a disk with the metadata, from websites, from remote shell accounts, or from Source Code Management (SCM) systems like `cvs/subversion/git`.

Fetchers are usually triggered by entries in `SRC_URI`. You can find information about the options and formats of entries for specific fetchers in the BitBake manual located in the `bitbake/doc/manual` directory of the Source Directory.

One useful feature for certain Source Code Manager (SCM) fetchers is the ability to "auto-update" when the upstream SCM changes version. Since this ability requires certain functionality from the SCM, not all systems support it. Currently Subversion, Bazaar and to a limited extent, Git support the ability to "auto-update". This feature works using the `SRCREV` variable. See the "Using an External SCM" section in the Yocto Project Development Manual for more information.

Chapter 7. Classes

Table of Contents

Class files are used to abstract common functionality and share it amongst multiple `.bb` files. Any Metadata usually found in a `.bb` file 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.

In most cases inheriting the class is enough to enable its features, although for some classes you might need to set variables or override some of the default behavior.

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.

7.1. The base Class - base.bbclass

The base class is special in that every `.bb` file inherits it automatically. 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 `autotools.bbclass` or `package.bbclass`. The class also contains some commonly used functions such as `oe_runmake`.

7.2. Autotooled Packages - autotools.bbclass

Autotools (`autoconf`, `automake`, and `libtool`) bring standardization. This class defines a set of tasks (configure, compile etc.) that work for all Autotooled packages. It should usually be enough to define a few standard variables and then simply `inherit autotools`. This class can also work with software that emulates Autotools. For more information, see the "Autotooled Package" section in the Yocto Project Development Manual.

It's useful to have some idea of how the tasks defined by this class 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` variable.

- `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 a destination directory option, which takes its value from the standard `DESTDIR` variable.

7.3. Alternatives - `update-alternatives.bbclass`

This 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.bbclass` 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. 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`.

Note

You can use the `update-alternatives` command directly in your recipes. However, this class simplifies things in most cases.

7.4. Initscripts - `update-rc.d.bbclass`

This 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.

7.5. Binary Configuration Scripts - `binconfig.bbclass`

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, BitBake installs such scripts into the `sysroots/` directory. BitBake also changes all paths to point into the `sysroots/` directory so all builds that use the script will use the correct directories for the cross compiling layout.

7.6. Debian Renaming - `debian.bbclass`

This class renames packages so that they follow the Debian naming policy (i.e. `eglibc` becomes `libc6` and `eglibc-devel` becomes `libc6-dev`.)

7.7. Pkg-config - `pkgconfig.bbclass`

`pkg-config` provides a standard way to get header and library information. 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`, this class no longer has to manipulate the files.

7.8. Archiving Sources - `archive*.bbclass`

Many software licenses require that source code and other materials be released with the binaries. To help with that task, the following classes are provided:

- `archive-original-sources.bbclass`
- `archive-patched-sources.bbclass`
- `archive-configured-sources.bbclass`
- `archiver.bbclass`

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 Manual.

7.9. Perl Modules - `cpan.bbclass`

Recipes for Perl modules are simple. These recipes usually only need to point to the source's archive and then inherit the proper `.bbclass` 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.

7.10. Python Extensions - `distutils.bbclass`

Recipes for Python extensions are simple. These recipes usually only need to point to the source's archive and then inherit the proper `.bbclass` file. 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 `distutils`-based `.bbclasse` files in their recipes.
- Extensions that use `distutils`-based build systems require `distutils.bbclass` in their recipes.

7.11. Developer Shell - `devshell.bbclass`

This class adds the `devshell` task. Distribution policy dictates whether to include this class. See the "Using a Development Shell" section in the Yocto Project Development Manual for more information about using `devshell`.

7.12. Package Groups - `packagegroup.bbclass`

This 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 Manual.

Previously, this class was named `task.bbclass`.

7.13. Packaging - `package*.bbclass`

The packaging classes add support for generating packages from a build's output. The core generic functionality is in `package.bbclass`. The code specific to particular package types is contained in various sub-classes such as `package_deb.bbclass`, `package_ipk.bbclass`, and `package_rpm.bbclass`. Most users will want one or more of these classes.

You can control the list of resulting package formats by using the PACKAGE_CLASSES variable defined in the local.conf configuration file, which is located in the conf folder of the Source 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 Smart, you can install packages from the feed while you are running the image on the target (i.e. runtime installation of packages). For information on how to set up this repository, see the "Setting Up Runtime Package Management" in the Yocto Project Development Manual.

The package 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 decision on package manager, 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 Berkley 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

7.14. Building Kernels - `kernel.bbclass`

This 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 module.bbclass.

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.

Various other classes are used by the kernel and module classes internally including kernel-arch.bbclass, module_strip.bbclass, module-base.bbclass, and linux-kernel-base.bbclass.

7.15. Creating Images - `image.bbclass` and `rootfs*.bbclass`

These classes add support for creating images in several formats. First, the root filesystem is created from packages using one of the rootfs_*.bbclass files (depending on the package format used) and then the image is 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.

7.16. Host System Sanity Checks - `sanity.bbclass`

This 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.

7.17. Generated Output Quality Assurance Checks - `insane.bbclass`

This class adds a step to the package generation process that sanity checks the packages 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. You use the `WARN_QA` variable to specify tests for which you want to generate a warning message on failure. You use the `ERROR_QA` variable to specify tests for which you want to generate an error message on failure.

The following list shows the tests you can list with the `WARN_QA` and `ERROR_QA` variables:

- ***`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.
- ***`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.
- ***`rpaths:`*** Checks for rpaths in the binaries that contain build system paths such as `TMPDIR`. If this test fails, bad `-rpath` options are being passed to the linker commands and your binaries have potential security issues.
- ***`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.
- ***`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.
- ***`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 don't match the type since there would be an incompatibility. Sometimes software, like bootloaders, might need to bypass this check.
- ***`debug-deps:`*** Checks that `-dbg` packages only depend on other `-dbg` packages and not on any other types of packages, which would cause a packaging bug.
- ***`dev-deps:`*** Checks that `-dev` packages only depend on other `-dev` packages and not on any other types of packages, which would be a packaging bug.
- ***`pkgconfig:`*** Checks `.pc` files for any `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.
- ***`textrel:`*** Checks for ELF binaries that contain relocations in their `.text` sections, which can result in a performance impact at runtime.
- ***`pkgvarcheck:`*** 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.
- ***`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.
- ***`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`.

- ***staticdev:*** Checks for static library files (`*.a`) in non-`staticdev` packages.
- ***la:*** Checks `.la` files for any `TMPDIR` paths. Any `.la` file containing these paths is incorrect since `libtool` adds the correct sysroot prefix when using the files automatically itself.
- ***desktop:*** Runs the `desktop-file-validate` program against any `.desktop` files to validate their contents against the specification for `.desktop` files.

Note

You can 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.

7.18. Removing Work Files During the Build - `rm_work.bbclass`

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 eglibc"
```

7.19. Autotools Configuration Data Cache - `siteinfo.bbclass`

Autotools 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.

Because this class is included from `base.bbclass`, it is always active.

7.20. Adding Users - `useradd.bbclass`

If you have packages that install files that are owned by custom users or groups, you can use this class to specify those packages and associate the users and groups with those packages. 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 for more information on how to use this class.

7.21. Using External Source - externalsrc.bbclass

You can use this class to build software from source code that is external to the OpenEmbedded build system. In other words, your source code resides in an external tree outside of the Yocto Project. Building software from an external source tree means that the normal fetch, unpack, and patch process is not used.

To use the class, you need to define the S variable to point to the directory that contains the source files. You also need to have your recipe inherit the externalsrc.bbclass class.

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 the glossary entries for the WORKDIR, BPN, PV, S, and B for more information.

You can build object files in the external tree by setting the B variable equal to "${S}". However, this practice does not work well if you use the source for more than one variant (i.e., "natives" such as quilt-native, or "crosses" such as gcc-cross). So, be sure there are no "native", "cross", or "multilib" variants of the recipe.

If you do want to build different variants of a recipe, you can use the BBCLASSEXTEND variable. When you do, the B variable must support the recipe's ability to build variants in different working directories. Most Autotools-based recipes support separating these directories. The OpenEmbedded build system defaults to using separate directories for gcc and some kernel recipes. Alternatively, you can make sure that separate recipes exist that each use the BBCLASSEXTEND variable to build each variant. The separate recipes can inherit a single target recipe.

For information on how to use this class, see the "Building Software from an External Source" section in the Yocto Project Development Manual.

7.22. Other Classes

Thus far, this chapter has discussed only the most useful and important classes. However, other classes exist within the meta/classes directory in the Source Directory. You can examine the .bbclass files directly for more information.

Chapter 8. Images

The OpenEmbedded build process supports several types of 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) components is only supported for minimal and base images. Furthermore, if you are going to build an image using non-GPLv3 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 = "GPLv3"
```

From within the `poky` Git repository, use the following command to list the supported images:

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

These recipes reside in the `meta/recipes-core/images`, `meta/recipes-extended/images`, `meta/recipes-graphics/images`, `meta/recipes-qt/images`, `meta/recipes-rt/images`, `meta/recipes-sato/images`, and `meta-skeleton/recipes-multilib/images` directories within the Source Directory. Although the recipe names are somewhat explanatory, here is a list that describes them:

- ***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-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.
- ***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-basic*:** 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.
- ***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.
- ***core-image-lsb-sdk*:** A `core-image-lsb` that includes everything in meta-toolchain but also includes development headers and libraries to form a complete standalone SDK. This image is suitable for development using the target.
- ***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-gtk-directfb*:** An image that uses `gtk+` over `directfb` instead of X11. In order to build, this image requires specific distro configuration that enables `gtk` over `directfb`.
- ***core-image-x11*:** A very basic X11 image with a terminal.
- ***qt4e-demo-image*:** An image that launches into the demo application for the embedded (not based on X11) version of Qt.
- ***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 `meta-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 meta-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-multilib-example:*** An example image that includes a `lib32` version of Bash into an otherwise standard `sato` image. The image assumes a "lib32" multilib has been enabled in the your configuration.

Tip

From the Yocto Project release 1.1 onwards, `-live` and `-directdisk` images have been replaced by a "live" option in `IMAGE_FSTYPES` that will work with any image to produce an image file that can be copied directly to a CD or USB device and run as is. To build a live image, simply add "live" to `IMAGE_FSTYPES` within the `local.conf` file or wherever appropriate and then build the desired image as normal.

Chapter 9. Reference: Features

Table of Contents

This chapter provides a reference of shipped machine and distro features you can include as part of the image, a reference on image types you can build, 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 in the `poky.conf` distribution configuration file. 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 $HOME/poky
$ git grep 'contains.*MACHINE_FEATURES.*<feature>'
```

9.1. Distro

The items below are features you can use with `DISTRO_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 `do_configure` for a particular recipe.

This list only represents features as shipped with the Yocto Project metadata:

- ***alsa:*** Include ALSA support (OSS compatibility kernel modules installed if available).
- ***bluetooth:*** Include bluetooth support (integrated BT only)
- ***cramfs:*** Include CramFS 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
- ***irda:*** Include Irda support
- ***keyboard:*** Include keyboard support (e.g. keymaps will be loaded during boot).
- ***nfs:*** Include NFS client support (for mounting NFS exports on device)
- ***pci:*** Include PCI bus support
- ***pcmcia:*** Include PCMCIA/CompactFlash support
- ***ppp:*** Include PPP dialup support
- ***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)

9.2. Machine

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 `do_configure` 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
- ***ext2:*** Hardware HDD or Microdrive
- ***irda:*** Hardware has Irda support
- ***keyboard:*** Hardware has a keyboard
- ***pci:*** Hardware has a PCI bus
- ***pcmcia:*** Hardware has PCMCIA or CompactFlash sockets
- ***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
- ***wifi:*** Hardware has integrated WiFi

9.3. Images

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.

Current list of IMAGE_FEATURES contains the following:

- ***dbg-pkgs:*** Installs debug symbol packages for all packages installed in a given image.
- ***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.
- ***nfs-server:*** Installs an NFS server.
- ***read-only-fsroot:*** Creates an image whose root filesystem is read-only. See the "Creating a Read-Only Root Filesystem" section in the Yocto Project Development 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.
- ***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.
- ***staticdev-pkgs:*** Installs static development packages (i.e. static libraries containing *.a files) for all packages installed in a given image.
- ***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 Manual. For information on tracing and profiling, see the Yocto Project Profiling and Tracing Manual.
- ***tools-profile:*** Installs profiling tools such as oprofile, exmap, and LTTng. For general information on user-space tools, see the "User-Space Tools" section in the Yocto Project Application Developer's Guide.
- ***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.

9.4. 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 10. Variables Glossary

Table of Contents

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 H I K L M O P R S T W

A

ALLOW_EMPTY

Specifies if an output package should still be produced 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 runtime hard-requirement on the existence of the package.

Like all package-controlling variables, you must always use them in conjunction with a package name override. Here is an example:

```
ALLOW_EMPTY_${PN} = "1"
```

AUTHOR

The email address used to contact the original author or authors in order to send patches and forward bugs.

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}"
```

B

B

The Build Directory. The OpenEmbedded build system places generated objects into the Build Directory during a recipe's build process. By default, this directory is the same as the S directory:

```
B = "${WORKDIR}/${BPN}/{PV}/"
```

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

A list of packages not to install despite being recommended by a recipe. Support for this variable exists only when using the IPK packaging backend.

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" in the `local.conf` file in the Build Directory as follows:

```
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 (
      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,${TMPDIR},1G,100K WARN,${SS
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>,<di

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).

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 "mulitlibs" 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>"
```

BBMASK

Prevents BitBake from processing recipes and recipe append files. Use the `BBMASK` variable from within the `conf/local.conf` file found in the Build Directory.

You can use the `BBMASK` variable to "hide" these `.bb` and `.bbappend` files. BitBake ignores any recipe or recipe append files that match the expression. It is as if BitBake does not see them at all. Consequently, matching files are not parsed or otherwise used by BitBake.

The value you provide is passed to Python's regular expression compiler. The expression is compared against the full paths to the files. For complete syntax information, see Python's documentation at http://docs.python.org/release/2.3/lib/re-syntax.html.

The following example uses a complete regular expression to tell BitBake to ignore all recipe and recipe append files in the `/meta-ti/recipes-misc/` directory:

```
BBMASK = "/meta-ti/recipes-misc/"
```

If you want to mask out multiple directories or recipes, use the vertical bar to separate the regular expression fragments. This next example masks out multiple directories and individual recipes:

```
BBMASK = "meta-ti/recipes-misc/|meta-ti/recipes-ti/pa
BBMASK .= "|.*meta-oe/recipes-support/"
BBMASK .= "|.*openldap"
BBMASK .= "|.*opencv"
BBMASK .= "|.*lzma"
```

Notice how the vertical bar is used to append the fragments.

Note

When specifying a directory name, use the trailing slash character to ensure you match just that directory name.

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.

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.

BBFILES
List of recipe files used by BitBake to build software.

BBPATH
Used by BitBake to locate `.bbclass` and configuration files. This variable is analogous to the `PATH` variable.

BBINCLUDELOGS
Variable that controls how BitBake displays logs on build failure.

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 \
  "

BBLAYERS_NON_REMOVABLE ?= " \
  /home/scottrif/poky/meta \
  /home/scottrif/poky/meta-yocto \
  "
```

This example enables four layers, one of which is a custom, user-defined layer named `meta-mykernel`.

BBLAYERS_NON_REMOVABLE

Lists core layers that cannot be removed from the `bblayers.conf` file. In order for BitBake to build your image, your `bblayers.conf` file must include the `meta` and `meta-yocto` core layers. Here is an example that shows these two layers listed in the `BBLAYERS_NON_REMOVABLE` statement:

```
BBLAYERS = " \
  /home/scottrif/poky/meta \
  /home/scottrif/poky/meta-yocto \
  /home/scottrif/poky/meta-yocto-bsp \
  /home/scottrif/poky/meta-mykernel \
  "

BBLAYERS_NON_REMOVABLE ?= " \
  /home/scottrif/poky/meta \
  /home/scottrif/poky/meta-yocto \
  "
```

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

The bare name of the recipe. This variable is a version of the `PN` variable but removes common suffixes such as "-native" and "-cross" as well as removes common prefixes such as multilib's "lib64-" and "lib32-". The exact list of suffixes removed is specified by the `SPECIAL_PKGSUFFIX` variable. The exact list of prefixes removed is specified by the `MLPREFIX` variable. Prefixes are removed for `multilib` and `nativesdk` cases.

BUILDDIR

Points to the location of the Build Directory. You can define this directory indirectly through the `oe-init-build-env` 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 `BUILDDIR` defaults to `build` in the current directory.

C

CFLAGS

Flags passed to the C compiler for the target system. This variable evaluates to the same as `TARGET_CFLAGS`.

COMBINED_FEATURES

A set of features common between `MACHINE_FEATURES` and `DISTRO_FEATURES`. See the glossary descriptions for these variables for more information.

COMPATIBLE_HOST

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 HOST_SYS. 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.

COMPATIBLE_MACHINE

A regular expression that resolves to one or more target machines with which a recipe is compatible. The regular expression is matched against MACHINEOVERRIDES. 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.

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_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`.

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.

D

D
The destination directory.

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 -g".

DEFAULT_PREFERENCE
Specifies a weak bias for recipe selection priority.

The most common usage of this is 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 the 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). The system ensures that all the dependencies listed have been built and have their contents in the appropriate sysroots before the recipe's configure task is executed.

DESCRIPTION
The package description used by package managers. If not set, DESCRIPTION takes the value of the SUMMARY variable.

DESTDIR
the destination directory.

DISTRO
The short name of the distribution. This variable corresponds to a file with the extension .conf located in a conf/distro directory within the Metadata that contains the distribution configuration. The value must not contain spaces, and is typically all lower-case.

If the variable is blank, a set of default configuration will be used, which is specified within meta/conf/distro /defaultsetup.conf.

DISTRO_EXTRA_RDEPENDS
Specifies a list of distro-specific packages to add to all images. This variable takes affect through packagegroup-base so the variable only really applies to the more full-featured images that include packagegroup-base. 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 .conf file.

DISTRO_EXTRA_RRECOMMENDS
: 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.

DISTRO_FEATURES
: The features enabled for the distribution. For a list of supported features that ship with the Yocto Project, see the "Distro" section.

DISTRO_FEATURES_BACKFILL
: Features to be added to `DISTRO_FEATURES` if not also present in `DISTRO_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 distro features are being backfilled for all distro configurations. See the Feature backfilling section for more information.

DISTRO_FEATURES_BACKFILL_CONSIDERED
: Features from `DISTRO_FEATURES_BACKFILL` that should not be backfilled (i.e. added to `DISTRO_FEATURES`) during the build. See the "Feature Backfilling" section for more information.

DISTRO_NAME
: The long name of the distribution.

DISTRO_PN_ALIAS
: Alias names used for the recipe in various Linux distributions.

  See the "Handling a Package Name Alias" section in the Yocto Project Development Manual for more information.

DISTRO_VERSION
: the version of the distribution.

DISTROOVERRIDES
: This variable lists overrides specific to the current distribution. By default, the variable list includes the value of the `DISTRO` variable. You can extend the variable to apply any variable overrides you want as part of the distribution and are not already in `OVERRIDES` through some other means.

DL_DIR
: The central download directory used by the build process to store downloads. You can set this directory by defining the `DL_DIR` variable in the `/conf/local.conf` file. This directory is self-maintaining and you should not have to touch it. By default, the directory is `downloads` in the Build Directory.

```
#DL_DIR ?= "${TOPDIR}/downloads"
```

  To specify a different download directory, simply uncomment the line and provide your directory.

  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.

E

ENABLE_BINARY_LOCALE_GENERATION

Variable that controls which locales for `eglibc` are generated during the build (useful if the target device has 64Mbytes of RAM or less).

EXTENDPE

Used with file and pathnames to create a prefix for a recipe's version based on the recipe's `PE` value. If `PE` is set and greater than zero for a recipe, `EXTENDPE` becomes that value (e.g if `PE` is equal to "1" then `EXTENDPE` becomes "1_"). If a recipe's `PE` is not set (the default) or is equal to zero, `EXTENDPE` becomes "".

See the `STAMP` variable for an example.

EXTRA_IMAGE_FEATURES

The list of additional features to include in an image. Typically, you configure this variable in your `local.conf` file, which is found in the Build Directory. Although you can use this variable from within a recipe, best practices dictate that you do not.

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 package
             including symbol information for debugging ar
             profiling.

"debug-tweaks" - Makes an image suitable for development.
                 For example, ssh root access has a blank
                 password.  You should remove this feature
                 before you produce a production image.

"dev-pkgs" - Adds -dev packages for all installed package
             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 Manual for more
                     information

"tools-debug" - Adds debugging tools such as gdb and
                strace.

"tools-profile" - Adds profiling tools such as oprofile,
                  exmap, lttng and valgrind (x86 only).

"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 "Images" 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 Manual.

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.

EXTRA_OECMAKE

Additional cmake options.

EXTRA_OECONF

Additional configure script options.

EXTRA_OEMAKE

Additional GNU make options.

F

FILES

The list of directories or files that are placed in packages.

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 identifies the files you want included as part of the resulting package. Here is an example:

```
FILES_${PN} += "${bindir}/mydir1/ ${bindir}/mydir2/myfi
```

Note

When specifying paths as part of the FILES 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.

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.

FILESEXTRAPATHS

Extends the search path the OpenEmbedded build system uses when looking for files and patches as it processes recipes and append files. The directories BitBake uses when it processes recipes are defined by the FILESPATH variable, and can be extended using FILESEXTRAPATHS.

Best practices dictate that you accomplish this by using the variable from within a .bbappend file and that you prepend paths as follows:

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

In the above example, the build system 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.

Here is a final example that specifically adds three paths:

```
FILESEXTRAPATHS_prepend := "path_1:path_2:path_3:"
```

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.

Note

Be sure to use the immediate expansion (:=) operator and include the trailing separating colon character.

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.

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},
   "${FILE_DIRNAME}/${BPN}", "${FILE_DIRNAME}/files"]
```

Note

Do not hand-edit the `FILESPATH` 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.

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 distros 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`.

FULL_OPTIMIZATION

The options to pass in `TARGET_CFLAGS` and `CFLAGS` when compiling an optimized system. This variable defaults to "-fexpensive-optimizations -fomit-frame-pointer -frename-registers -O2".

H

HOMEPAGE

Website where more information about the software the recipe is building can be found.

HOST_SYS

Specifies the system, including the architecture and the operating system, for with the build is occurring in the context of the current recipe. The OpenEmbedded build system automatically sets this variable. You do not need to set the variable yourself.

Here are 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".

I

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 "Images" section.

For 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 Manual.

IMAGE_FSTYPES

Formats of root filesystem images that you want to have created.

IMAGE_INSTALL

Specifies the packages to install into an image. The `IMAGE_INSTALL` variable is a mechanism for an image recipe and you should use it 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 exist, such as `core-image.bbclass`, that can take `IMAGE_FEATURES` lists and turn these into auto-generated entries in `IMAGE_INSTALL` in addition to its default contents.

Using `IMAGE_INSTALL` with the `+=` operator from the `/conf/local.conf` file or from within an image recipe is not recommended as it can cause ordering issues. Since `core-image.bbclass` sets `IMAGE_INSTALL` to a default value using the `?=` operator, using a `+=` operation against `IMAGE_INSTALL` will result in unexpected behavior when used in `/conf/local.conf`. Furthermore, the same operation from with 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.

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.

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).

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 `IMAGE_ROOTFS_EXTRA_SPACE` the variable.

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"
```

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) + xsp

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 modification

  xspace = IMAGE_ROOTFS_EXTRA_SPACE
```

See the `IMAGE_OVERHEAD_FACTOR` and `IMAGE_ROOTFS_EXTRA_SPACE` variables for related information.

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 = '
recipes-graphics/xorg-font/encodings_1.0.4.bb:PR = "${INC_
recipes-graphics/xorg-font/font-util_1.3.0.bb:PR = "${INC_
recipes-graphics/xorg-font/font-alias_1.0.3.bb:PR = "${IN(
```

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.

INHIBIT_PACKAGE_STRIP

If set to "1", causes the build to not strip binaries in resulting packages.

INHERIT

Causes the named class to be inherited at this point during parsing. The variable is only valid in configuration files.

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_NAME

The filename of the initscript as installed to `${etcdir}/init.d`.

This variable is used in recipes when using `update-rc.d.bbclass`. The variable is Mandatory.

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 is mandatory and is used in recipes when using `update-rc.d.bbclass`.

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 "Generated Output Quality Assurance Checks - `insane.bbclass`" section for a list of the valid QA checks you can specify using this variable.

K

KARCH

Defines the kernel architecture used when assembling the configuration. Architectures supported for this release are:

```
powerpc
arm
i386
mips
powerpc
x86_64
```

You define the `KARCH` variable in the BSP Descriptions.

KBRANCH

A regular expression used by the build process to explicitly identify the kernel branch that is validated, patched and configured during a build. The `KBRANCH` variable is optional. You can use it to trigger checks to ensure the exact kernel branch you want is being used by the build process.

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 Yocto Project kernel that is based on the Linux 3.4 kernel, the kernel recipe file is the `meta/recipes-kernel/linux/linux-yocto_3.4.bb` file. Following is the default value for `KBRANCH` and the default override for the architectures the Yocto Project supports:

```
KBRANCH_DEFAULT = "standard/base"
KBRANCH = "${KBRANCH_DEFAULT}"
```

This branch exists in the `linux-yocto-3.4` kernel Git repository http://git.yoctoproject.org/cgit.cgi/linux-yocto-3.4/refs/heads.

This variable is also used from the kernel's append file to identify the kernel branch specific to a particular machine or target hardware. The kernel's append file is located in the BSP layer for a given machine. For example, the kernel append file for the Crown Bay BSP is in the `meta-intel` Git repository and is named `meta-crownbay/recipes-kernel/linux/linux-yocto_3.4.bbappend`. Here are the related statements from the append file:

```
COMPATIBLE_MACHINE_crownbay = "crownbay"
KMACHINE_crownbay  = "crownbay"
KBRANCH_crownbay  = "standard/crownbay"

COMPATIBLE_MACHINE_crownbay-noemgd = "crownbay-noemgd
KMACHINE_crownbay-noemgd  = "crownbay"
KBRANCH_crownbay-noemgd  = "standard/crownbay"
```

The `KBRANCH_*` statements identify the kernel branch to use when building for the Crown Bay BSP. In this case there are two identical statements: one for each type of Crown Bay machine.

KBRANCH_DEFAULT

Defines the Linux kernel source repository's default branch used to build the Linux kernel. The `KBRANCH_DEFAULT` value is the default value for `KBRANCH`. Unless you specify otherwise, `KBRANCH_DEFAULT` initializes to "master".

KERNEL_EXTRA_ARGS

Specifies additional `make` command-line arguments the OpenEmbedded build system passes on when compiling the kernel.

KERNEL_FEATURES

Includes additional metadata from the Yocto Project kernel Git repository. 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 to further add metadata for all 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 adds `netfilter` to all the Yocto Project kernels and adds sound support to the `qemux86` machine:

```
# Add netfilter to all linux-yocto kernels
KERNEL_FEATURES="features/netfilter"

# Add sound support to the qemux86 machine
KERNEL_FEATURES_append_qemux86=" cfg/sound"
```

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.

KERNEL_PATH

The location of the kernel sources. This variable is set to the value of the STAGING_KERNEL_DIR within the `module.bbclass` class. For information on how this variable is used, see the "Incorporating Out-of-Tree Modules" section.

The KERNEL_SRC variable is identical to the `KERNEL_PATH` variable.

KERNEL_SRC

The location of the kernel sources. This variable is set to the value of the STAGING_KERNEL_DIR within the `module.bbclass` class. For information on how this variable is used, see the "Incorporating Out-of-Tree Modules" section.

The KERNEL_PATH variable is identical to the `KERNEL_SRC` variable.

KFEATURE_DESCRIPTION

Provides a short description of a configuration fragment. You use this variable in the `.scc` file that describes a configuration fragment file. Here is the variable used in a file named `smp.scc` 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 `qemuarm` goes by a different name in the Linux Yocto kernel. The kernel understands that machine as `arm_versatile926ejs`. For cases like these, the `KMACHINE` variable maps the kernel machine name to the OpenEmbedded build system machine name.

Kernel machine names are initially defined in the Yocto Linux Kernel's `meta` branch. From the `meta` branch, look in the `meta/cfg/kernel-cache/bsp/<bsp_name>/<bsp-name>-<kernel-type>.scc` file. For example, from the `meta` branch in the `linux-yocto-3.0` kernel, the `meta/cfg/kernel-cache/bsp/cedartrail/cedartrail-standard.scc` file has the following:

```
define KMACHINE cedartrail
define KTYPE standard
define KARCH i386

include ktypes/standard
branch cedartrail

include cedartrail.scc
```

You can see that the kernel understands the machine name for the Cedar Trail Board Support Package (BSP) as `cedartrail`.

If you look in the Cedar Trail BSP layer in the `meta-intel` Source Repositories at `meta-cedartrail/recipes-kernel/linux/linux-yocto_3.0.bbappend`, you will find the following statements among others:

```
COMPATIBLE_MACHINE_cedartrail = "cedartrail"
KMACHINE_cedartrail  = "cedartrail"
KBRANCH_cedartrail  = "yocto/standard/cedartrail"
KERNEL_FEATURES_append_cedartrail += "bsp/cedartrail/
KERNEL_FEATURES_append_cedartrail += "cfg/efi-ext.sc

COMPATIBLE_MACHINE_cedartrail-nopvr = "cedartrail"
KMACHINE_cedartrail-nopvr  = "cedartrail"
```

```
KBRANCH_cedartrail-nopvr  = "yocto/standard/cedartra:
KERNEL_FEATURES_append_cedartrail-nopvr += " cfg/smp
```

The KMACHINE statements in the kernel's append file make sure that the OpenEmbedded build system and the Yocto Linux kernel understand the same machine names.

This append file uses two KMACHINE statements. The first is not really necessary but does ensure that the machine known to the OpenEmbedded build system as cedartrail maps to the machine in the kernel also known as cedartrail:

```
KMACHINE_cedartrail  = "cedartrail"
```

The second statement is a good example of why the KMACHINE variable is needed. In this example, the OpenEmbedded build system uses the cedartrail-nopvr machine name to refer to the Cedar Trail BSP that does not support the proprietary PowerVR driver. The kernel, however, uses the machine name cedartrail. Thus, the append file must map the cedartrail-nopvr machine name to the kernel's cedartrail name:

```
KMACHINE_cedartrail-nopvr  = "cedartrail"
```

BSPs that ship with the Yocto Project release provide all mappings between the Yocto Project kernel machine names and the OpenEmbedded machine names. Be sure to use the KMACHINE if you create a BSP and the machine name you use is different than that used in the kernel.

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

LAYERDEPENDS

Lists the layers that this recipe depends upon, separated by spaces. 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). An error will be produced if any dependency is missing or the version numbers do not match exactly (if specified). 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.

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).

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

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.

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.

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_TYI
```

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 ?= "qemumips"
MACHINE ?= "qemuppc"
MACHINE ?= "qemux86"
MACHINE ?= "qemux86-64"
MACHINE ?= "atom-pc"
MACHINE ?= "beagleboard"
MACHINE ?= "mpc8315e-rdb"
MACHINE ?= "routerstationpro"
```

The last four 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_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
```

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-basic` 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-basic` 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-examplew:
```

MACHINE_FEATURES

Specifies the list of hardware features the MACHINE supports. For example, including the "bluetooth" feature causes the `bluez` bluetooth daemon to be built and added to the image. It also causes the `connman` recipe to look at `MACHINE_FEATURES` and when it finds "bluetooth" there it enables the bluetooth support in ConnMan.

For a list of features supported by the Yocto Project as shipped, see the "Machine" 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 "Feature backfilling" section for more information.

MACHINEOVERRIDES

Lists overrides specific to the current machine. By default, this list includes the value of MACHINE. You can extend the list to apply variable overrides for classes of machines. For example, all QEMU emulated machines (e.g. qemuarm, qemux86, and so forth) include a common file named meta/conf/machine/include/qemu.inc that prepends MACHINEOVERRIDES with the following variable override:

```
MACHINEOVERRIDES =. "qemuall:"
```

Applying an override like qemuall affects all QEMU emulated machines elsewhere. Here is an example from the connman-conf recipe:

```
SRC_URI_append_qemuall = "file://wired.config \
                          file://wired-setup \
                         "
```

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-yocto Git repository.

MLPREFIX

Specifies a prefix has been added to PN to create a special version of a recipe or package, such as 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.

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.

MULTIMACH_TARGET_SYS

Separates files for different machines such that you can build for multiple target machines using the same output directories. See the STAMP variable for an example.

N

NATIVELSBSTRING

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.

O

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 Manual.

You can use the following values for the `OE_TERMINAL` variable:

```
auto
gnome
xfce
rxvt
screen
konsole
none
```

Note

Konsole support only works for KDE 3.x. Also, "auto" is the default behavior for `OE_TERMINAL`

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.

PACKAGE_BEFORE_PN

Enables easily adding packages to `PACKAGES` before `${PN}` so that the 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 Source Directory, specifies the package manager to use when packaging data. You can provide one or more arguments for the variable with the first argument being the package manager used to create images:

```
PACKAGE_CLASSES ?= "package_rpm package_deb package_:
```

For information on build performance effects as a result of the package manager use, see Packaging - `package*.bbclass` in this manual.

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).

PACKAGECONFIG

This variable provides a means of enabling or disabling features of a recipe on a per-recipe basis. The `PACKAGECONFIG` variable itself specifies a space-separated list of the features to enable. The features themselves are specified as flags on the `PACKAGECONFIG` variable. You can provide up to four arguments, which are separated by commas, to determine the behavior of each feature when it is enabled or disabled. You can omit any argument you like but must retain the separating commas. The arguments specify the following:

1. Extra arguments that should be added to the configure script argument list (`EXTRA_OECONF`) if the feature is enabled.
2. Extra arguments that should be added to `EXTRA_OECONF` if the feature is disabled.
3. Additional build dependencies (`DEPENDS`) that should be added if the feature is enabled.
4. Additional runtime dependencies (`RDEPENDS`) that should be added if the feature is enabled.

Consider the following example taken from the `librsvg` recipe. In this example the feature is `croco`, which has three arguments that determine the feature's behavior.

```
PACKAGECONFIG ??= "croco"
PACKAGECONFIG[croco] = "--with-croco,--without-croco
```

The `--with-croco` and `libcroco` arguments apply only if the feature is enabled. In this case, `--with-croco` is added to the configure script argument list and `libcroco` is added to `DEPENDS`. On the other hand, if the feature is disabled say through a `.bbappend` file in another layer, then the second argument `--without-croco` is added to the configure script rather than `--with-croco`.

PACKAGES

The list of packages to be created from the recipe. The default value is the following:

```
${PN}-dbg ${PN}-staticdev ${PN}-dev ${PN}-doc ${PN}-
```

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 `do_rootfs`.

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 Manual.

PARALLEL_MAKE

Specifies extra options that are passed to the make command during the compile tasks. This variable is usually in the form -j 4, where the number represents the maximum number of parallel threads make can run. If you development host supports multiple cores a good rule of thumb is to set this variable to twice the number of cores on the host.

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/). This variable is comprised of the following:

```
${PN}-${EXTENDPE}${PV}-${PR}
```

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.

PR

The revision of the recipe. The default value for this variable is "r0".

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-yocto 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.

PRINC

Causes the PR variable of .bbappend files to dynamically increment. This increment minimizes the impact of layer ordering.

In order to ensure multiple .bbappend files can co-exist, PRINC should be self referencing. This variable defaults to 0.

Following is an example that increments PR by two:

```
PRINC := "${@int(PRINC) + 2}"
```

It is advisable not to use strings such as ".= '.1'" with the variable because this usage is very sensitive to layer ordering. You should avoid explicit assignments as they cannot adequately represent multiple .bbappend files.

PROVIDES

A list of aliases that a recipe also provides. These aliases are useful for satisfying dependencies of other recipes during the build (as specified by DEPENDS).

Note

A recipe's own PN is implicitly already in its PROVIDES list.

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).

PE

the epoch of the recipe. The default value is "0". The field is used to make upgrades possible when the versioning scheme changes in some backwards incompatible way.

PREFERRED_PROVIDER

If multiple recipes provide an item, this variable determines which recipe should be given preference. 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. Here is an example:

```
PREFERRED_PROVIDER_virtual/xserver = "xserver-xf86"
```

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 to the PV accordingly for precedence. You can use the "%" character as a wildcard to match any number of characters, which can be useful when specifying versions that contain long revision number that could potentially change. Here are two examples:

```
PREFERRED_VERSION_python = "2.6.6"
PREFERRED_VERSION_linux-yocto = "3.0+git%"
```

R

RCONFLICTS

The list of packages that conflict with another package. Note that the package will not be installed if the 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'
```

RDEPENDS

Lists a package's run-time 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.

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. 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.

The package name you attach to the `RDEPENDS` variable must appear as it would in the `PACKAGES` namespace before any renaming of the output package by classes like `debian.bbclass`.

In many cases you do not need to explicitly add run-time dependencies using `RDEPENDS` since some automatic handling occurs:

- ***shlibdeps***: If a run-time package contains a shared library (`.so`), the build processes the library in order to determine other libraries to which it is dynamically linked. The build process adds these libraries to `RDEPENDS` when creating the run-time package.
- ***pcdeps***: If the package ships a `pkg-config` information file, the build process uses this file to add items to the `RDEPENDS` variable to create the run-time packages.

RM_WORK_EXCLUDE

With `rm_work` enabled, this variable specifies a list of recipes whose work directories should not be removed. See the "Removing Work Files During the Build - `rm_work.bbclass`" section for more details.

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`).

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"
```

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.

The OpenEmbedded build process automatically installs the list of packages as part of the built package. However, you can remove them later if you want. If, during the build, a package from the list cannot be found, the build process continues without an 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 like debian.bbclass.

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"
```

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 package source code resides. This location is within the working directory (WORKDIR), which is not static. The unpacked source location depends on the package name (PN) and package version (PV) as follows:

```
${WORKDIR}/${PN}/${PV}
```

As an example, assume a Source Directory top-level folder named `poky` and a default Build Directory at `poky/build`. In this case, the working directory the build system uses to build the `db` package is the following:

```
~/poky/build/tmp/work/qemux86-poky-linux/db/5.1.19-r3/db
```

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 `NATIVELSBSTRING` 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.

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
```

SECTION

The section in which packages should be categorized. Package management utilities can make use of this variable.

SELECTED_OPTIMIZATION

The variable takes the value of `FULL_OPTIMIZATION` unless `DEBUG_BUILD` = "1". In this case the value of `DEBUG_OPTIMIZATION` is used.

SERIAL_CONSOLE

The speed and device for the serial port used to attach the serial console. This variable is given to the kernel as the "console" parameter and after booting occurs `getty` is started on that port so remote login is possible.

SIGGEN_EXCLUDERECIPES_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.

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 this example, `intone` depends on `mplayer2`.

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.

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.

SITEINFO_BITS

Specifies the number of bits for the target system CPU. The value should be either "32" or "64".

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`.

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.

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:

- ***`file://`* -** Fetches files, which are usually files shipped with the Metadata, from the local machine. The path is relative to the `FILESPATH` 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.
- ***`svk://`*** **-** Fetches files from an SVK revision control 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.

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.

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 `mindate`.
- ***`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.
- ***subdir*** **-** Places the file (or extracts its contents) into the specified subdirectory of `WORKDIR`. 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.

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

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).

SRCPV

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 = "1.5.0+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 wish to build a fixed revision and you wish 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.

SSTATE_DIR

The directory for the shared state cache.

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 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.

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 \n
file://.* file:///some/local/dir/sstate/PATH"
```

STAGING_KERNEL_DIR

The directory with kernel headers that are required to build out-of-tree modules.

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. Currently, the default assignment for STAMP as set in the meta/conf/bitbake.conf file is:

```
STAMP = "${STAMPS_DIR}/${MULTIMACH_TARGET_SYS}/${PN}/
```

See STAMPS_DIR, MULTIMACH_TARGET_SYS, PN, EXTENDPE, PV, and PR for related variable information.

STAMPS_DIR

Specifies the base directory in which the OpenEmbedded build system places stamps. The default directory is ${TMPDIR}/stamps.

SUMMARY

The short (72 characters or less) summary of the binary package for packaging systems such as opkg, rpm or dpkg. By default, SUMMARY is used to define the DESCRIPTION variable if DESCRIPTION is not set in the recipe.

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.

T

T

This variable points to a directory were 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 architecture of the device being built. The OpenEmbedded build system supports the following architectures:

```
arm
mips
ppc
x86
x86-64
```

TARGET_CFLAGS

Flags passed to the C compiler for the target system. This variable evaluates to the same as CFLAGS.

TARGET_FPU

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.

TARGET_OS

Specifies the target's operating system. The variable can be set to "linux" for `eglibc`-based systems and to "linux-uclibc" for `uclibc`. For ARM/EABI targets, there are also "linux-gnueabi" and "linux-uclibc-gnueabi" values possible.

TCLIBC

Specifies which variant of the GNU standard C library (`libc`) to use during the build process. This variable replaces `POKYLIBC`, which is no longer supported.

You can select `eglibc` or `uclibc`.

Note

This release of the Yocto Project does not support the `glibc` implementation of `libc`.

TCMODE

The toolchain selector. This variable replaces `POKYMODE`, which is no longer supported.

The `TCMODE` variable selects the external toolchain built using the OpenEmbedded build system or a few supported combinations of the upstream GCC or CodeSourcery Labs toolchain. The variable identifies the `tcmode-*` files used in the `meta/conf/distro/include` directory, which is found in the Source Directory.

By default, `TCMODE` is set to "default", which chooses the `tcmode-default.inc` file. The variable is similar to `TCLIBC`, which controls the variant of the GNU standard C library (`libc`) used during the build process: `eglibc` or `uclibc`.

THISDIR

The directory in which the file BitBake is currently parsing is located. Do not manually set this variable.

TMPDIR

This variable is the temporary directory the OpenEmbedded build system uses when it does its work building images. By default, the `TMPDIR` variable is named `tmp` within the Build Directory.

If you want to establish this directory in a location other than the default, you can uncomment the following statement in the `conf/local.conf` file in the Source Directory:

```
#TMPDIR = "${TOPDIR}/tmp"
```

TOPDIR

This variable is the Build Directory. BitBake automatically sets this variable. The OpenEmbedded build system uses the Build Directory when building images.

W

WORKDIR

The pathname of the working directory in which the OpenEmbedded build system builds a recipe. This directory is located within the `TMPDIR` directory structure and changes as different packages are built.

The actual `WORKDIR` directory depends on several things:

- The temporary directory - `TMPDIR`
- The package architecture - `PACKAGE_ARCH`
- The target machine - `MACHINE`
- The target operating system - `TARGET_OS`
- The recipe name - `PN`
- The recipe version - `PV`
- The recipe revision - `PR`

For packages that are not dependent on a particular machine, `WORKDIR` is defined as follows:

```
${TMPDIR}/work/${PACKAGE_ARCH}-poky-${TARGET_OS}/${PN}/$·
```

As an example, assume a Source Directory top-level folder name `poky` and a default Build Directory at `poky/build`. In this case, the working directory the build system uses to build the `v86d` package is the following:

```
     ~/poky/build/tmp/work/qemux86-poky-linux/v86d/01.9-r(
```

For packages that are dependent on a particular machine, `WORKDIR` is defined slightly different:

```
${TMPDIR}/work/${MACHINE}-poky-${TARGET_OS}/${PN}/${PV}-$
```

As an example, again assume a Source Directory top-level folder named `poky` and a default Build Directory at `poky/build`. In this case, the working directory the build system uses to build the `acl` recipe, which is being built for a MIPS-based device, is the following:

```
     ~/poky/build/tmp/work/mips-poky-linux/acl/2.2.51-r2
```

Chapter 11. Variable Context

Table of Contents

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.

11.1. Configuration

The following subsections provide lists of variables whose context is configuration: distribution, machine, and local.

11.1.1. 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

11.1.2. Machine

This section lists variables whose configuration context is the machine.

- TARGET_ARCH
- SERIAL_CONSOLE
- PACKAGE_EXTRA_ARCHS
- IMAGE_FSTYPES
- MACHINE_FEATURES
- MACHINE_EXTRA_RDEPENDS
- MACHINE_EXTRA_RRECOMMENDS
- MACHINE_ESSENTIAL_EXTRA_RDEPENDS
- MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS

11.1.3. 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

11.2. Recipes

The following subsections provide lists of variables whose context is recipes: required, dependencies, path, and extra build information.

11.2.1. 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.

11.2.2. Dependencies

This section lists variables that define recipe dependencies.

- DEPENDS
- RDEPENDS
- RRECOMMENDS
- RCONFLICTS
- RREPLACES

11.2.3. Paths

This section lists variables that define recipe paths.

- WORKDIR
- S
- FILES

11.2.4. Extra Build Information

This section lists variables that define extra build information for recipes.

- EXTRA_OECMAKE
- EXTRA_OECONF
- EXTRA_OEMAKE
- PACKAGES
- DEFAULT_PREFERENCE

Chapter 12. FAQ

12.1. 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.

12.2. I only have Python 2.4 or 2.5 but BitBake requires Python 2.6 or 2.7. Can I still use the Yocto Project?

You can use a stand-alone tarball to provide Python 2.6. You can find pre-built 32 and 64-bit versions of Python 2.6 at the following locations:

- 32-bit tarball
- 64-bit tarball

These tarballs are self-contained with all required libraries and should work on most Linux systems. To use the tarballs extract them into the root directory and run the appropriate command:

```
$ export PATH=/opt/poky/sysroots/i586-pokysdk-linux/usr/bin/:$PATH
$ export PATH=/opt/poky/sysroots/x86_64-pokysdk-linux/usr/bin/:$PATH
```

Once you run the command, BitBake uses Python 2.6.

12.3. 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 an autobuilder, which provides continuous build and integration tests.

12.4. 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 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.

12.5. 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.

12.6. 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.

12.7. 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 add a package, see the section "Writing a Recipe to Add a Package to Your Image" in the Yocto Project Development Manual.

12.8. 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.

12.9. What is GNOME Mobile and what is the difference between GNOME Mobile and GNOME?

GNOME Mobile is a subset of the GNOME platform targeted at mobile and embedded devices. The the main difference between GNOME Mobile and standard GNOME is that desktop-orientated libraries have been removed, along with deprecated libraries, creating a much smaller footprint.

12.10. 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.

12.11. How do I make the Yocto Project work in RHEL/CentOS?

To get the Yocto Project working under RHEL/CentOS 5.1 you need to first install some required packages. The standard CentOS packages needed are:

- "Development tools" (selected during installation)
- `texi2html`
- `compat-gcc-34`

On top of these, you need the following external packages:

- `python-sqlite2` from DAG repository
- `help2man` from Karan repository

Once these packages are installed, the OpenEmbedded build system will be able to build standard images. However, there might be a problem with the QEMU emulator segfaulting. You can either disable the generation of binary locales by setting `ENABLE_BINARY_LOCALE_GENERATION` to "0" or by removing the `linux-2.6-execshield.patch` from the kernel and rebuilding it since that is the patch that causes the problems with QEMU.

Note

For information on distributions that the Yocto Project uses during validation, see the Distribution Support Wiki page.

For notes about using the Yocto Project on a RHEL 4-based host, see the Building on RHEL4 Wiki page.

12.12. I see lots of 404 responses for files on `http://www.yoctoproject.org/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.

12.13. 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.

12.14. 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 in your home directory. Here are some example settings:

```
http_proxy = http://proxy.yoyodyne.com:18023/
ftp_proxy = http://proxy.yoyodyne.com:18023/
```

The Yocto Project also includes a `site.conf.sample` file that shows how to configure CVS and Git proxy servers if needed.

12.15. What’s the difference between `foo` and `foo-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.

12.16. 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.

12.17. 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" and "Maintaining Open Source License Compliance During Your Product's Lifecycle" sections, both of which are in the Yocto Project Development Manual.

12.18. How do I disable the cursor on my touchscreen device?

You need to create a form factor file as described in the "Miscellaneous 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
```

12.19. 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 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
```

12.20. 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`.

12.21. 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.

12.22. 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" (i.e. `tcmode-default.inc`). However, other patterns are accepted. In particular, "external-*" refers to external toolchains of which there are some basic examples included in the OpenEmbedded Core (`meta`). You can use your own custom toolchain definition in your own layer (or as defined in the `local.conf` file) at the location `conf/distro/include/tcmode-*.inc`.

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`, `libstdcc++`, any locales, and `libc`. An example is the `external-sourcery-toolchain.bb`, which is located in `meta/recipes-core/meta/` within the Source Directory.

For information on installing and using cross-development toolchains, see the "Installing the ADT and Toolchains" section in the Yocto Project Application Developer's Guide.

12.23. 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.

12.24. 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 source the `oe-init-build-env` setup script. 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.

Chapter 13. Contributing to the Yocto Project

Table of Contents

13.1. 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. To find out how to download source code, see the "Yocto Project Release" section in the Yocto Project Development Manual.

13.2. Tracking Bugs

If you find problems with the Yocto Project, you should report them using the Bugzilla application at http://bugzilla.yoctoproject.org.

13.3. 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.linuxtogo.org/cgi-bin/mailman/listinfo/openembedded-core - Discussion mailing list about OpenEmbedded-Core (the core metadata).
- http://lists.linuxtogo.org/cgi-bin/mailman/listinfo/openembedded-devel - Discussion mailing list about OpenEmbedded.
- http://lists.linuxtogo.org/cgi-bin/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.

13.4. Internet Relay Chat (IRC)

Two IRC channels on freenode are available for the Yocto Project and Poky discussions:

- `#yocto`
- `#poky`

13.5. Links

Here is a list of resources you will find helpful:

- ***The Yocto Project website:*** The home site for the Yocto Project.
- ***Intel Corporation:*** The company who acquired OpenedHand in 2008 and began development on the Yocto Project.
- ***OpenEmbedded:*** The upstream, generic, embedded distribution used as the basis for the build system in the Yocto Project. Poky derives from and contributes back to the OpenEmbedded project.
- ***BitBake:*** The tool used to process metadata.
- ***BitBake User Manual:*** A comprehensive guide to the BitBake tool. You can find the BitBake User Manual in the `bitbake/doc/manual` directory, which is found in the Source Directory.
- ***QEMU:*** An open source machine emulator and virtualizer.

13.6. 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 find out who is the maintainer for areas of code, see the "How to Submit a Change" section in the Yocto Project Development Manual.