The Yocto Project Reference Manual

Richard Purdie, Linux Foundation
<richard.purdie@linuxfoundation.org>

by Richard Purdie

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 [http://www.yoctoproject.org/docs/1.1.1/poky-ref-manual/poky-ref-manual.html] on the Yocto Project [http://www.yoctoproject.org] website. For the latest version of this manual, see the manual on the website.

Table of Contents

Chapter 1. Introduction

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 Poky build tool 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 [http://www.yoctoproject.org/docs/1.1.1/yocto-project-qs/yocto-project-qs.html]. For task-based information using the Yocto Project, see The Yocto Project Development Manual [http://www.yoctoproject.org/docs/1.1.1//dev-manual/dev-manual.html]. You can also find lots of information on the Yocto Project on the Yocto Project website [http://www.yoctoproject.org].

1.2. Documentation Overview

This reference manual consists of the following:

- Using the Yocto Project: This chapter provides an overview of the components that make up the Yocto Project followed by information about debugging images created in the Yocto Project.
- Extending the Yocto Project: This chapter provides information about how to extend and customize the Yocto Project along with advice on how to manage these changes.
- Technical Details: This chapter describes fundamental Yocto Project components as well as an explanation behind how the Yocto Project uses shared state (sstate) cache to speed build time.
- Board Support Packages (BSP) - Developer's Guide: This chapter describes the example filesystem layout for BSP development and the click-through licensing scheme.
- Platform Development With the Yocto Project: This chapter describes application development, debugging, and profiling using the Yocto Project.
- Reference: Directory Structure: This appendix describes the directory structure of the Yocto Project files. The Yocto Project files represent the file structure or Git repository created as a result of setting up the Yocto Project on your host development system.
- Reference: BitBake: This appendix provides an overview of the BitBake tool and its role within the Yocto Project.
- Reference: Classes: This appendix describes the classes used in the Yocto Project.
- Reference: Images: This appendix describes the standard images that the Yocto Project supports.
- Reference: Features: This appendix describes mechanisms for creating distribution, machine, and image features during the build process using the Yocto Project.
- Reference: Variables Glossary: This appendix presents most Yocto Project variables. Entries describe the function of the variable and how to apply them.
- Reference: Variable Context: This appendix provides variable locality or context.
- Reference: FAQ: This appendix provides answers for commonly asked questions in the Yocto Project development environment.
- Reference: Contributing to the Yocto Project: This appendix provides guidance on how you can contribute back to the Yocto Project.

1.3. System Requirements

For system Yocto Project system requirements, see the What You Need and How You Get It [http://www.yoctoproject.org/docs/1.1.1/yocto-project-qs/yocto-project-qs.html#resources] section in the Yocto Project Quick Start [http://www.yoctoproject.org/docs/1.1.1/yocto-project-qs/yocto-project-qs.html].

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 tarballs, and experimental builds.
- Yocto Project Website: You can find releases of the Yocto Project and supported BSPs at the Yocto Project website [http://www.yoctoproject.org]. 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 copy of the Yocto Project files. You can get these files by downloading a Yocto Project release tarball and unpacking it, or by establishing a Git repository of the files. For information on both these methods, see Getting Setup [http://www.yoctoproject.org/docs/1.1.1/dev-manual/dev-manual.html#getting-setup] section in The Yocto Project Development Manual [http://www.yoctoproject.org/docs/1.1.1/dev-manual/dev-manual.html].

Chapter 2. Using the Yocto Project

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

2.1. Running a Build

You can find general information on how to build an image using the Yocto Project in the Building an Image [http://www.yoctoproject.org/docs/1.1.1/yocto-project-qs/yocto-project-qs.html#building-image] section of the Yocto Project Quick Start [http://www.yoctoproject.org/docs/1.1.1/yocto-project-qs/yocto-project-qs.html]. This section provides a summary of the build process and provides information for less obvious aspects of the build process.

2.1.1. Build Overview

The first thing you need to do is set up the Yocto Project 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 Yocto Project uses for the build. 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 oe-init-build-env for more information on this script.

Once the Yocto Project 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 Yocto Project files. 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 Yocto Project supports, see the 'Reference: Images' appendix.

> **Note**
> Building an image without GNU Public License Version 3 (GPLv3) components is only supported for minimal and base images. See 'Reference: Images' 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 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 Yocto Project 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 [http://www.yoctoproject.org/docs/1.1.1/yocto-project-qs/yocto-project-qs.html#using-pre-built] section in the Yocto Project Quick Start [http://www.yoctoproject.org/docs/1.1.1/yocto-project-qs/yocto-project-qs.html]. For information about how to install these images, see the documentation for your particular board/machine.

2.3. Debugging Build Failures

The exact method for debugging Yocto Project 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 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.

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 as follows:

```
$ 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 `matchbox-desktop` and then recompiles it. 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
```

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` 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. 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.

2.3.6. Variables

The `-e` option dumps the resulting environment for either the configuration (no package specified) or for a specific package when specified; or `-b recipename` to show the environment from parsing a single recipe file only.

2.3.7. 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` directory of the Yocto Project files.

2.3.7.1. 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 figureing out the tasklist")
}
```

2.3.7.2. 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"
}
```

2.3.8. 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).

Chapter 3. Common Tasks

This chapter describes standard tasks such as adding new software packages, extending or customizing images or porting the Yocto Project to new hardware (adding a new machine). The chapter also describes ways to modify package source code, combine multiple versions of library files into a single image, track license changes, and handle a package name alias. Finally, the chapter contains advice about how to make changes to the Yocto Project to achieve the best results.

3.1. Adding a Package

To add a package into the Yocto Project you need to write a recipe for it. Writing a recipe means creating a `.bb` file that sets some variables. For information on variables that are useful for recipes and for information about recipe naming issues, see the Required section for recipe variables.

Before writing a recipe from scratch, it is often useful to check whether someone else has written one already. OpenEmbedded is a good place to look as it has a wider scope and range of packages. Because the Yocto Project aims to be compatible with OpenEmbedded, most recipes you find there should work in Yocto Project.

For new packages, the simplest way to add a recipe is to base it on a similar pre-existing recipe. The sections that follow provide some examples that show how to add standard types of packages.

3.1.1. Single .c File Package (Hello World!)

Building an application from a single file that is stored locally (e.g. under `files/`) requires a recipe that has the file listed in the `SRC_URI` variable. Additionally, you need to manually write the `do_compile` and `do_install` tasks. The `S` variable defines the directory containing the source code, which is set to `WORKDIR` in this case - the directory BitBake uses for the build.

```
DESCRIPTION = "Simple helloworld application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"
PR = "r0"

SRC_URI = "file://helloworld.c"

S = "${WORKDIR}"

do_compile() {
 ${CC} helloworld.c -o helloworld
}

do_install() {
 install -d ${D}${bindir}
 install -m 0755 helloworld ${D}${bindir}
}
```

By default, the `helloworld`, `helloworld-dbg`, and `helloworld-dev` packages are built. For information on how to customize the packaging process, see the "Splitting an Application into Multiple Packages" section.

3.1.2. Autotooled Package

Applications that use Autotools such as `autoconf` and `automake` require a recipe that has a source archive listed in `SRC_URI` and also inherits Autotools, which instructs BitBake to use the `autotools.bbclass` file, which contains the definitions of all the steps needed to build an Autotool-based application. The result of the build is automatically packaged. And, if the application uses NLS for localization, packages with local information are generated (one package per language). Following is one example: (`hello_2.3.bb`)

```
DESCRIPTION = "GNU Helloworld application"
SECTION = "examples"
LICENSE = "GPLv2+"
LIC_FILES_CHKSUM = "file://COPYING;md5=751419260aa954499f7abaabaa882bbe"
PR = "r0"

SRC_URI = "${GNU_MIRROR}/hello/hello-${PV}.tar.gz"

inherit autotools gettext
```

The variable LIC_FILES_CHKSUM is used to track source license changes as described in the Track License Change section. You can quickly create Autotool-based recipes in a manner similar to the previous example.

3.1.3. Makefile-Based Package

Applications that use GNU make also require a recipe that has the source archive listed in SRC_URI. You do not need to add a do_compile step since by default BitBake starts the make command to compile the application. If you need additional make options you should store them in the EXTRA_OEMAKE variable. BitBake passes these options into the make GNU invocation. Note that a do_install task is still required. Otherwise BitBake runs an empty do_install task by default.

Some applications might require extra parameters to be passed to the compiler. For example, the application might need an additional header path. You can accomplish this by adding to the CFLAGS variable. The following example shows this:

```
CFLAGS_prepend = "-I ${S}/include "
```

In the following example, mtd-utils is a makefile-based package:

```
DESCRIPTION = "Tools for managing memory technology devices."
SECTION = "base"
DEPENDS = "zlib lzo e2fsprogs util-linux"
HOMEPAGE = "http://www.linux-mtd.infradead.org/"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=0636e73ff0215e8d672dc4c32c317bb3 \
                    file://include/common.h;beginline=1;endline=17;md5=ba05b07912a44ea2bf81c

SRC_URI = "git://git.infradead.org/mtd-utils.git;protocol=git;tag=v${PV}"

S = "${WORKDIR}/git/"

EXTRA_OEMAKE = "'CC=${CC}' 'CFLAGS=${CFLAGS} -I${S}/include -DWITHOUT_XATTR' \
                'BUILDDIR=${S}'"

do_install () {
        oe_runmake install DESTDIR=${D} SBINDIR=${sbindir} MANDIR=${mandir} \
                           INCLUDEDIR=${includedir}
        install -d ${D}${includedir}/mtd/
        for f in ${S}/include/mtd/*.h; do
                install -m 0644 $f ${D}${includedir}/mtd/
        done
}
```

3.1.4. Splitting an Application into Multiple Packages

You can use the variables PACKAGES and FILES to split an application into multiple packages.

Following is an example that uses the libXpm recipe. By default, this recipe generates a single package that contains the library along with a few binaries. You can modify the recipe to split the binaries into separate packages:

```
require xorg-lib-common.inc

DESCRIPTION = "X11 Pixmap library"
LICENSE = "X-BSD"
LIC_FILES_CHKSUM = "file://COPYING;md5=3e07763d16963c3af12db271a31abaa5"
DEPENDS += "libxext libsm libxt"
PR = "r3"
PE = "1"

XORG_PN = "libXpm"

PACKAGES =+ "sxpm cxpm"
FILES_cxpm = "${bindir}/cxpm"
FILES_sxpm = "${bindir}/sxpm"
```

In the previous example, we want to ship the `sxpm` and `cxpm` binaries in separate packages. Since `bindir` would be packaged into the main PN package by default, we prepend the PACKAGES variable so additional package names are added to the start of list. This results in the extra `FILES_*` variables then containing information that define which files and directories go into which packages. Files included by earlier packages are skipped by latter packages. Thus, the main PN package does not include the above listed files.

3.1.5. Including Static Library Files

If you are building a library and the library offers static linking, you can control which static library files (`*.a` files) get included in the built library.

The PACKAGES and `FILES_*` variables in the `meta/conf/bitbake.conf` configuration file define how files installed by the `do_install` task are packaged. By default, the PACKAGES variable contains `${PN}-staticdev`, which includes all static library files.

Note

Previously released versions of the Yocto Project defined the static library files through `${PN}-dev`.

Following, is part of the BitBake configuration file. You can see where the static library files are defined:

```
PACKAGES = "${PN}-dbg ${PN} ${PN}-doc ${PN}-dev ${PN}-staticdev ${PN}-locale"
PACKAGES_DYNAMIC = "${PN}-locale-*"
FILES = ""

FILES_${PN} = "${bindir}/* ${sbindir}/* ${libexecdir}/* ${libdir}/lib*${SOLIBS} \
            ${sysconfdir} ${sharedstatedir} ${localstatedir} \
            ${base_bindir}/* ${base_sbindir}/* \
            ${base_libdir}/*${SOLIBS} \
            ${datadir}/${BPN} ${libdir}/${BPN}/* \
            ${datadir}/pixmaps ${datadir}/applications \
            ${datadir}/idl ${datadir}/omf ${datadir}/sounds \
            ${libdir}/bonobo/servers"

FILES_${PN}-doc = "${docdir} ${mandir} ${infodir} ${datadir}/gtk-doc \
            ${datadir}/gnome/help"
SECTION_${PN}-doc = "doc"

FILES_${PN}-dev = "${includedir} ${libdir}/lib*${SOLIBSDEV} ${libdir}/*.la \
                ${libdir}/*.o ${libdir}/pkgconfig ${datadir}/pkgconfig \
                ${datadir}/aclocal ${base_libdir}/*.o"
SECTION_${PN}-dev = "devel"
ALLOW_EMPTY_${PN}-dev = "1"
RDEPENDS_${PN}-dev = "${PN} (= ${EXTENDPKGV})"

FILES_${PN}-staticdev = "${libdir}/*.a ${base_libdir}/*.a"
```

```
SECTION_${PN}-staticdev = "devel"
RDEPENDS_${PN}-staticdev = "${PN}-dev (= ${EXTENDPKGV})"
```

3.1.6. Post Install Scripts

To add a post-installation script to a package, add a `pkg_postinst_PACKAGENAME()` function to the `.bb` file and use `PACKAGENAME` as the name of the package you want to attach to the `postinst` script. Normally `PN` can be used, which automatically expands to PACKAGENAME. A post-installation function has the following structure:

```
pkg_postinst_PACKAGENAME () {
#!/bin/sh -e
# Commands to carry out
}
```

The script defined in the post-installation function is called when the root filesystem is created. If the script succeeds, the package is marked as installed. If the script fails, the package is marked as unpacked and the script is executed when the image boots again.

Sometimes it is necessary for the execution of a post-installation script to be delayed until the first boot. For example, the script might need to be executed on the device itself. To delay script execution until boot time, use the following structure in the post-installation script:

```
pkg_postinst_PACKAGENAME () {
#!/bin/sh -e
if [ x"$D" = "x" ]; then
     # Actions to carry out on the device go here
else
     exit 1
fi
}
```

The previous example delays execution until the image boots again because the `D` variable points to the directory containing the image when the root filesystem is created at build time but is unset when executed on the first boot.

3.2. Customizing Images

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

3.2.1. Customizing Images Using Custom .bb Files

One way to get additional software into an image is to create a custom image. The following example shows the form for the two lines you need:

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

inherit core-image
```

By creating a custom image, a developer has total control over the contents of the image. It is important to use the correct names of packages in the `IMAGE_INSTALL` variable. You must use the OpenEmbedded notation and not the Debian notation for the names (e.g. `eglibc-dev` instead of `libc6-dev`).

The other method for creating a custom image is to modify an existing image. For example, if a developer wants to add `strace` into the `core-image-sato` image, they can use the following recipe:

```
require core-image-sato.bb

IMAGE_INSTALL += "strace"
```

3.2.2. Customizing Images Using Custom Tasks

For complex custom images, the best approach is to create a custom task package that is used to build the image or images. A good example of a tasks package is `meta/recipes-sato/tasks/task-poky.bb`. The `PACKAGES` variable lists the task packages to build along with the complementary `-dbg` and `-dev` packages. For each package added, you can use `RDEPENDS` and `RRECOMMENDS` entries to provide a list of packages the parent task package should contain. Following is an example:

```
DESCRIPTION = "My Custom Tasks"

PACKAGES = "\
    task-custom-apps \
    task-custom-apps-dbg \
    task-custom-apps-dev \
    task-custom-tools \
    task-custom-tools-dbg \
    task-custom-tools-dev \
    "

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

RDEPENDS_task-custom-tools = "\
    oprofile \
    oprofileui-server \
    lttng-control \
    lttng-viewer"

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

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

3.2.3. Customizing Images Using Custom **IMAGE_FEATURES** and **EXTRA_IMAGE_FEATURES**

Ultimately users might want to add extra image features to the set used by Yocto Project with the `IMAGE_FEATURES` variable. To create these features, the best reference is `meta/classes/core-image.bbclass`, which shows how the Yocto Project achieves this. In summary, the file looks at the contents of the `IMAGE_FEATURES` variable and then maps that into a set of tasks or packages. Based on this information the `IMAGE_INSTALL` variable is generated automatically. Users can add extra features by extending the class or creating a custom class for use with specialized image `.bb` files. You can also add more features by configuring the `EXTRA_IMAGE_FEATURES` variable in the `local.conf` file found in the Yocto Project files located in the build directory.

The Yocto Project ships with two SSH servers you can use in your images: Dropbear and OpenSSH. Dropbear is a minimal SSH server appropriate for resource-constrained environments, while OpenSSH is a well-known standard SSH server implementation. By default, the `core-image-sato` image is configured to use Dropbear. The `core-image-basic` and `core-image-lsb` images both include OpenSSH. To change these defaults, edit the `IMAGE_FEATURES` variable so that it sets the image you are working with to include `ssh-server-dropbear` or `ssh-server-openssh`.

3.2.4. Customizing Images Using **local.conf**

It is possible to customize image contents by using variables used by distribution maintainers in the `local.conf` found in the Yocto Project build directory. This method only allows the addition of packages and is not recommended.

For example, to add the `strace` package into the image, you would add this package to the `local.conf` file:

```
DISTRO_EXTRA_RDEPENDS += "strace"
```

However, since the `DISTRO_EXTRA_RDEPENDS` variable is for distribution maintainers, adding packages using this method is not as simple as adding them using a custom `.bb` file. Using the `local.conf` file method could result in some packages needing to be recreated. For example, if packages were previously created and the image was rebuilt, then the packages would need to be recreated.

Cleaning `task-*` packages are required because they use the `DISTRO_EXTRA_RDEPENDS` variable. You do not have to build them by hand because Yocto Project images depend on the packages they contain. This means dependencies are automatically built when the image builds. For this reason we do not use the `rebuild` task. In this case the `rebuild` task does not care about dependencies - it only rebuilds the specified package.

```
$ bitbake -c clean task-boot task-base task-poky
$ bitbake core-image-sato
```

3.3. Porting the Yocto Project to a New Machine

Adding a new machine to the Yocto Project is a straightforward process. This section provides information that gives you an idea of the changes you must make. The information covers adding machines similar to those the Yocto Project already supports. Although well within the capabilities of the Yocto Project, adding a totally new architecture might require changes to `gcc/eglibc` and to the site information, which is beyond the scope of this manual.

For a complete example that shows how to add a new machine to the Yocto Project, see the BSP Development Example [http://www.yoctoproject.org/docs/1.1.1/dev-manual/dev-manual.html#dev-manual-bsp-appendix] in Appendix A of The Yocto Project Development Manual [http://www.yoctoproject.org/docs/1.1.1/dev-manual/dev-manual.html].

3.3.1. Adding the Machine Configuration File

To add a machine configuration you need to add a `.conf` file with details of the device being added to the `conf/machine/` file. The name of the file determines the name the Yocto Project uses to reference the new machine.

The most important variables to set in this file are as follows:

- `TARGET_ARCH` (e.g. "arm")
- `PREFERRED_PROVIDER_`virtual/kernel (see below)
- `MACHINE_FEATURES` (e.g. "kernel26 apm screen wifi")

You might also need these variables:

- `SERIAL_CONSOLE` (e.g. "115200 ttyS0")
- `KERNEL_IMAGETYPE` (e.g. "zImage")
- `IMAGE_FSTYPES` (e.g. "tar.gz jffs2")

You can find full details on these variables in the reference section. You can leverage many existing machine `.conf` files from `meta/conf/machine/`.

3.3.2. Adding a Kernel for the Machine

The Yocto Project needs to be able to build a kernel for the machine. You need to either create a new kernel recipe for this machine, or extend an existing recipe. You can find several kernel examples in the Yocto Project file's `meta/recipes-kernel/linux` directory that you can use as references.

If you are creating a new recipe, normal recipe-writing rules apply for setting up a `SRC_URI`. Thus, you need to specify any necessary patches and set S to point at the source code. You need to create a `configure` task that configures the unpacked kernel with a defconfig. You can do this by using a `make defconfig` command or, more commonly, by copying in a suitable `defconfig` file and and then running `make oldconfig`. By making use of `inherit kernel` and potentially some of the `linux-*.inc` files, most other functionality is centralized and the the defaults of the class normally work well.

If you are extending an existing kernel, it is usually a matter of adding a suitable defconfig file. The file needs to be added into a location similar to defconfig files used for other machines in a given kernel. A possible way to do this is by listing the file in the `SRC_URI` and adding the machine to the expression in `COMPATIBLE_MACHINE`:

```
COMPATIBLE_MACHINE = '(qemux86|qemumips)'
```

3.3.3. Adding a Formfactor Configuration File

A formfactor configuration file provides information about the target hardware for which the Yocto Project is building and information that the Yocto Project cannot obtain from other sources such as the kernel. Some examples of information contained in a formfactor configuration file include framebuffer orientation, whether or not the system has a keyboard, the positioning of the keyboard in relation to the screen, and the screen resolution.

The Yocto Project uses reasonable defaults in most cases, but if customization is necessary you need to create a `machconfig` file in the Yocto Project file's `meta/recipes-bsp/formfactor/files` directory. This directory contains directories for specific machines such as `qemuarm` and `qemux86`. For information about the settings available and the defaults, see the `meta/recipes-bsp/formfactor/files/config` file found in the same area. Following is an example for qemuarm:

```
HAVE_TOUCHSCREEN=1
HAVE_KEYBOARD=1

DISPLAY_CAN_ROTATE=0
DISPLAY_ORIENTATION=0
#DISPLAY_WIDTH_PIXELS=640
#DISPLAY_HEIGHT_PIXELS=480
#DISPLAY_BPP=16
DISPLAY_DPI=150
DISPLAY_SUBPIXEL_ORDER=vrgb
```

3.4. Modifying Package Source Code

Although the Yocto Project is usually used to build software, you can use it to modify software.

During a build, source is available in the `WORKDIR` directory. The actual location depends on the type of package and the architecture of the target device. For a standard recipe not related to `MACHINE`, the location is `tmp/work/PACKAGE_ARCH-poky-TARGET_OS/PN-PV-PR/`. For target device-dependent packages, you should use the `MACHINE` variable instead of `PACKAGE_ARCH` in the directory name.

Tip

Be sure the package recipe sets the S variable to something other than the standard `WORKDIR/PN-PV/` value.

After building a package, you can modify the package source code without problems. The easiest way to test your changes is by calling the `compile` task as shown in the following example:

```
$ bitbake -c compile -f NAME_OF_PACKAGE
```

The `-f` or `--force` option forces re-execution of the specified task. You can call other tasks this way as well. But note that all the modifications in `WORKDIR` are gone once you execute `-c clean` for a package.

3.5. Modifying Package Source Code with Quilt

By default Poky uses Quilt [http://savannah.nongnu.org/projects/quilt] to manage patches in the `do_patch` task. This is a powerful tool that you can use to track all modifications to package sources.

Before modifying source code, it is important to notify Quilt so it can track the changes into the new patch file:

```
$ quilt new NAME-OF-PATCH.patch
```

After notifying Quilt, add all modified files into that patch:

```
$ quilt add file1 file2 file3
```

You can now start editing. Once you are done editing, you need to use Quilt to generate the final patch that will contain all your modifications.

```
$ quilt refresh
```

You can find the resulting patch file in the `patches/` subdirectory of the source (S) directory. For future builds, you should copy the patch into the Yocto Project metadata and add it into the `SRC_URI` of a recipe. Here is an example:

```
SRC_URI += "file://NAME-OF-PATCH.patch"
```

Finally, don't forget to 'bump' the PR value in the same recipe since the resulting packages have changed.

3.6. Combining Multiple Versions of Library Files into One Image

The build system offers the ability to build libraries with different target optimizations or architecture formats and combine these together into one system image. You can link different binaries in the image against the different libraries as needed for specific use cases. This feature is called "Multilib."

An example would be where you have most of a system compiled in 32-bit mode using 32-bit libraries, but you have something large, like a database engine, that needs to be a 64-bit application and use 64-bit libraries. Multilib allows you to get the best of both 32-bit and 64-bit libraries.

While the Multilib feature is most commonly used for 32 and 64-bit differences, the approach the build system uses facilitates different target optimizations. You could compile some binaries to use one set of libraries and other binaries to use other different sets of libraries. The libraries could differ in architecture, compiler options, or other optimizations.

This section overviews the Multilib process only. For more details on how to implement Multilib, see the Multilib [https://wiki.yoctoproject.org/wiki/Multilib] wiki page.

3.6.1. Preparing to use Multilib

User-specific requirements drive the Multilib feature, Consequently, there is no one "out-of-the-box" configuration that likely exists to meet your needs.

In order to enable Multilib, you first need to ensure your recipe is extended to support multiple libraries. Many standard recipes are already extended and support multiple libraries. You can check in the `meta/conf/multilib.conf` configuration file in the Yocto Project files directory to see how this is done using the `BBCLASSEXTEND` variable. Eventually, all recipes will be covered and this list will be unneeded.

For the most part, the Multilib class extension works automatically to extend the package name from `${PN}` to `${MLPREFIX}${PN}`, where `MLPREFIX` is the particular multilib (e.g. "lib32-" or "lib64-"). Standard variables such as `DEPENDS`, `RDEPENDS`, `RPROVIDES`, `RRECOMMENDS`, `PACKAGES`, and `PACKAGES_DYNAMIC` are automatically extended by the system. If you are extending any manual code in the recipe, you can use the `${MLPREFIX}` variable to ensure those names are extended correctly. This automatic extension code resides in `multilib.bbclass`.

3.6.2. Using Multilib

After you have set up the recipes, you need to define the actual combination of multiple libraries you want to build. You accomplish this through your `local.conf` configuration file in the Yocto Project build directory. An example configuration would be as follows:

```
MACHINE = "qemux86-64"
require conf/multilib.conf
MULTILIBS = "multilib:lib32"
DEFAULTTUNE_virtclass-multilib-lib32 = "x86"
MULTILIB_IMAGE_INSTALL = "lib32-connman"
```

This example enables an additional library named `lib32` alongside the normal target packages. When combining these "lib32" alternatives, the example uses "x86" for tuning. For information on this particular tuning, see `meta/conf/machine/include/ia32/arch-ia32.inc`.

The example then includes `lib32-connman` in all the images, which illustrates one method of including a multiple library dependency. You can use a normal image build to include this dependency, for example:

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

You can also build Multilib packages specifically with a command like this:

```
$  bitbake lib32-connman
```

3.6.3. Additional Implementation Details

Different packaging systems have different levels of native Multilib support. For the RPM Package Management System, the following implementation details exist:

- A unique architecture is defined for the Multilib packages, along with creating a unique deploy folder under `tmp/deploy/rpm` in the Yocto Project build directory. For example, consider `lib32` in a `qemux86-64` image. The possible architectures in the system are "all", "qemux86_64", "lib32_qemux86_64", and "lib32_x86".
- The `${MLPREFIX}` variable is stripped from `${PN}` during RPM packaging. The naming for a normal RPM package and a Multilib RPM package in a `qemux86-64` system resolves to something similar to `bash-4.1-r2.x86_64.rpm` and `bash-4.1.r2.lib32_x86.rpm`, respectively.
- When installing a Multilib image, the RPM backend first installs the base image and then installs the Multilib libraries.

- The build system relies on RPM to resolve the identical files in the two (or more) Multilib packages.

For the IPK Package Management System, the following implementation details exist:

- The `${MLPREFIX}` is not stripped from `${PN}` during IPK packaging. The naming for a normal RPM package and a Multilib IPK package in a `qemux86-64` system resolves to something like `bash_4.1-r2.x86_64.ipk` and `lib32-bash_4.1-rw_x86.ipk`, respectively.
- The IPK deploy folder is not modified with `${MLPREFIX}` because packages with and without the Multilib feature can exist in the same folder due to the `${PN}` differences.
- IPK defines a sanity check for Multilib installation using certain rules for file comparison, overridden, etc.

3.7. Tracking License Changes

The license of an upstream project might change in the future. In order to prevent these changes going unnoticed, the Yocto Project provides a `LIC_FILES_CHKSUM` variable to track 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.

3.7.1. 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 Yocto Project 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;startline=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 this variable is mandatory for all recipes, unless the `LICENSE` variable is set to "CLOSED".

3.7.2. 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.

3.8. Handling a Package Name Alias

Sometimes a package name you are using might exist under an alias or as a similarly named package in a different distribution. The Yocto Project implements a distro_check task that automatically connects to major distributions and checks for these situations. If the package exists under a different name in a different distribution, you get a distro_check mismatch. You can resolve this problem by defining a per-distro recipe name alias using the DISTRO_PN_ALIAS variable.

Following is an example that shows how you specify the DISTRO_PN_ALIAS variable:

```
DISTRO_PN_ALIAS_pn-PACKAGENAME = "distro1=package_name_alias1 \
                                  distro2=package_name_alias2 \
                                  distro3=package_name_alias3 \
                                  ..."
```

If you have more than one distribution alias, separate them with a space. Note that the Yocto Project currently automatically checks the Fedora, OpenSuSE, Debian, Ubuntu, and Mandriva distributions for source package recipes without having to specify them using the DISTRO_PN_ALIAS variable. For example, the following command generates a report that lists the Linux distributions that include the sources for each of the Yocto Project recipes.

```
$ bitbake world -f -c distro_check
```

The results are stored in the build/tmp/log/distro_check-${DATETIME}.results file found in the Yocto Project files area.

3.9. Making and Maintaining Changes

Because the Yocto Project is extremely configurable and flexible, we recognize that developers will want to extend, configure or optimize it for their specific uses. To best keep pace with future Yocto Project changes, we recommend you make controlled changes to the Yocto Project.

The Yocto Project supports a "layers" concept. If you use layers properly, you can ease future upgrades and allow segregation between the Yocto Project core and a given developer's changes. The following section provides more advice on managing changes to the Yocto Project.

3.9.1. BitBake Layers

Often, developers want to extend the Yocto Project either by adding packages or by overriding files contained within the Yocto Project to add their own functionality. BitBake has a powerful mechanism called "layers", which provides a way to handle this extension in a fully supported and non-invasive fashion.

The Yocto Project files include several additional layers such as meta-rt and meta-yocto that demonstrate this functionality. The meta-rt layer is not enabled by default. However, the meta-yocto layer is.

To enable a layer, you simply add the layer's path to the BBLAYERS variable in your `bblayers.conf` file, which is found in the Yocto Project file's build directory. The following example shows how to enable the `meta-rt`:

```
LCONF_VERSION = "1"

 BBFILES ?= ""
 BBLAYERS = " \
  /path/to/poky/meta \
  /path/to/poky/meta-yocto \
  /path/to/poky/meta-rt \
  "
```

BitBake parses each `conf/layer.conf` file for each layer in BBLAYERS and adds the recipes, classes and configurations contained within the layer to the Yocto Project. To create your own layer, independent of the Yocto Project files, simply create a directory with a `conf/layer.conf` file and add the directory to your `bblayers.conf` file.

The `meta-yocto/conf/layer.conf` file demonstrates the required syntax:

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

# We have a packages directory, add to BBFILES
BBFILES := "${BBFILES} ${LAYERDIR}/recipes-*/*/*.bb \
            ${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "yocto"
BBFILE_PATTERN_yocto := "^${LAYERDIR}/"
BBFILE_PRIORITY_yocto = "5"
```

In the previous example, the recipes for the layers are added to BBFILES. The BBFILE_COLLECTIONS variable is then appended with the layer name. The BBFILE_PATTERN variable immediately expands with a regular expression used to match files from BBFILES into a particular layer, in this case by using the base pathname. The BBFILE_PRIORITY variable then assigns different priorities to the files in different layers. Applying priorities is useful in situations where the same package might appear in multiple layers and allows you to choose what layer should take precedence.

Note the use of the LAYERDIR variable with the immediate expansion operator. The LAYERDIR variable expands to the directory of the current layer and requires the immediate expansion operator so that BitBake does not wait to expand the variable when it's parsing a different directory.

BitBake can locate where other `.bbclass` and configuration files are applied through the BBPATH environment variable. For these cases, BitBake uses the first file with the matching name found in BBPATH. This is similar to the way the PATH variable is used for binaries. We recommend, therefore, that you use unique `.bbclass` and configuration file names in your custom layer.

We also recommend the following:

- Store custom layers in a Git repository that uses the `meta-prvt-XXXX` format.
- Clone the repository alongside other `meta` directories in the Yocto Project source files area.

Following these recommendations keeps your Yocto Project files area and its configuration entirely inside the Yocto Project's core base.

3.9.2. Committing Changes

Modifications to the Yocto Project are often managed under some kind of source revision control system. Because some simple practices can significantly improve usability, policy for committing changes is important. It helps to use a consistent documentation style when committing changes. The Yocto Project development team has found the following practices work well:

- The first line of the commit summarizes the change and begins with the name of the affected package or packages. However, not all changes apply to specific packages. Consequently, the prefix could also be a machine name or class name.
- The second part of the commit (if needed) is a longer more detailed description of the changes. Placing a blank line between the first and second parts helps with readability.

Following is an example commit:

```
bitbake/data.py: Add emit_func() and generate_dependencies() functions

These functions allow generation of dependency data between functions and
variables allowing moves to be made towards generating checksums and allowing
use of the dependency information in other parts of BitBake.

Signed-off-by: Richard Purdie richard.purdie@linuxfoundation.org
```

All commits should be self-contained such that they leave the metadata in a consistent state that builds both before and after the commit is made. Besides being a good practice to follow, it helps ensure autobuilder test results are valid.

3.9.3. Package Revision Incrementing

If a committed change results in changing the package output, then the value of the PR variable needs to be increased (or "bumped") as part of that commit. This means that for new recipes you must be sure to add the PR variable and set its initial value equal to "r0". Failing to define PR makes it easy to miss when you bump a package. Note that you can only use integer values following the "r" in the PR variable.

If you are sharing a common `.inc` file with multiple recipes, you can also use the `INC_PR` variable to ensure that the recipes sharing the `.inc` file are rebuilt when the `.inc` file itself is changed. The `.inc` file must set `INC_PR` (initially to "r0"), and all recipes referring to it should set PR to "$(INC_PR).0" initially, incrementing the last number when the recipe is changed. If the `.inc` file is changed then its `INC_PR` should be incremented.

When upgrading the version of a package, assuming the PV changes, the PR variable should be reset to "r0" (or "$(INC_PR).0" if you are using `INC_PR`).

Usually, version increases occur only to packages. However, if for some reason PV changes but does not increase, you can increase the PE variable (Package Epoch). The PE variable defaults to "0".

Version numbering strives to follow the Debian Version Field Policy Guidelines [http://www.debian.org/doc/debian-policy/ch-controlfields.html]. These guidelines define how versions are compared and what "increasing" a version means.

There are two reasons for following the previously mentioned guidelines. First, to ensure that when a developer updates and rebuilds, they get all the changes to the repository and do not have to remember to rebuild any sections. Second, to ensure that target users are able to upgrade their devices using package manager commands such as `opkg upgrade` (or similar commands for dpkg/apt or rpm-based systems).

The goal is to ensure the Yocto Project has packages that can be upgraded in all cases.

3.9.4. Using The Yocto Project in a Team Environment

It might not be immediately clear how you can use the Yocto Project in a team environment, or scale it for a large team of developers. The specifics of any situation determine the best solution. Granted that the Yocto Project offers immense flexibility regarding this, practices do exist that experience has shown work well.

The core component of any development effort with the Yocto Project is often an automated build and testing framework along with an image generation process. You can use these core components to check that the metadata can be built, highlight when commits break the build, and provide up-to-date images that allow developers to test the end result and use it as a base platform for further

development. Experience shows that buildbot is a good fit for this role. What works well is to configure buildbot to make two types of builds: incremental and full (from scratch). See the buildbot for the Yocto Project [http://www.yoctoproject.org:8010] for an example implementation that uses buildbot.

You can tie incremental builds to a commit hook that triggers the build each time a commit is made to the metadata. This practice results in useful acid tests that determine whether a given commit breaks the build in some serious way. Associating a build to a commit can catch a lot of simple errors. Furthermore, the tests are fast so developers can get quick feedback on changes.

Full builds build and test everything from the ground up. These types of builds usually happen at predetermined times like during the night when the machine load is low.

Most teams have many pieces of software undergoing active development at any given time. You can derive large benefits by putting these pieces under the control of a source control system that is compatible with the Yocto Project (i.e. Git or Subversion (SVN). You can then set the autobuilder to pull the latest revisions of the packages and test the latest commits by the builds. This practice quickly highlights issues. The Yocto Project easily supports testing configurations that use both a stable known good revision and a floating revision. The Yocto Project can also take just the changes from specific source control branches. This capability allows you to track and test specific changes.

Perhaps the hardest part of setting this up is defining the software project or the Yocto Project metadata policies that surround the different source control systems. Of course circumstances will be different in each case. However, this situation reveals one of the Yocto Project's advantages - the system itself does not force any particular policy on users, unlike a lot of build systems. The system allows the best policies to be chosen for the given circumstances.

3.9.5. Updating Existing Images

Often, rather than re-flashing a new image, you might wish to install updated packages into an existing running system. You can do this by first sharing the `tmp/deploy/ipk/` directory through a web server and then by changing `/etc/opkg/base-feeds.conf` to point at the shared server. Following is an example:

```
$ src/gz all http://www.mysite.com/somedir/deploy/ipk/all
$ src/gz armv7a http://www.mysite.com/somedir/deploy/ipk/armv7a
$ src/gz beagleboard http://www.mysite.com/somedir/deploy/ipk/beagleboard
```

Chapter 4. Technical Details

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

4.1. Yocto Project Components

The BitBake task executor together with various types of configuration files form the Yocto Project core. This section overviews the BitBake task executor and the configuration files 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: An abstraction of common build information (e.g. how to build a Linux kernel).
- Configuration Data: Defines machine-specific settings, policy decisions, etc. Configuration data acts as the glue to bind everything together.

For more information on data, see the Yocto Project Terms [http://www.yoctoproject.org/docs/1.1.1//dev-manual/dev-manual.html#yocto-project-terms] section in The Yocto Project Development Manual [http://www.yoctoproject.org/docs/1.1.1//dev-manual/dev-manual.html].

BitBake knows how to combine multiple data sources together and refers to each data source as a "layer".

Following are some brief details on these core components. For more detailed information on these components see the 'Reference: Directory Structure' appendix.

4.1.1. BitBake

BitBake is the tool at the heart of the Yocto Project 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 Yocto Project 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.

4.1.2. 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 Yocto Project (i.e. `.ipk` or `.deb` files), this document avoids using the term "package" when refering to recipes.

4.1.3. 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 Reference: Classes appendix provides details about common classes and how to use them.

4.1.4. Configuration

The configuration files (`.conf`) define various configuration variables that govern the Yocto Project 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 (`local.conf`, which is found in the Yocto Project files build directory).

4.2. Shared State Cache

By design, the Yocto Project build system builds everything from scratch unless BitBake can determine that parts don't 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 rebuiding things that don't 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 Yocto Project 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 don't 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.

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

4.2.1. 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.

4.2.2. 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/cross packages relocatable. The checksum therefore needs to exclude WORKDIR. The simplistic approach for excluding the worknig 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 inline 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.

While figuring out the dependencies and creating these checksums is good, what does the Yocto Project build system do with the checksum information? The build system uses a signature handler that is responsible for processing the checksum information. By default, there is a dummy "noop"

signature handler enabled in BitBake. This means that behaviour is unchanged from previous versions. OECore uses the "basic" signature handler through this setting in the `bitbake.conf` file:

```
BB_SIGNATURE_HANDLER ?= "basic"
```

Also within the BitBake configuration file, we can give BitBake some extra information to help it handle this information. 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"
BB_HASHTASK_WHITELIST += "(.*-cross$|.*-native$|.*-cross-initial$| \
    .*-cross-intermediate$|^virtual:native:.*|^virtual:nativesdk:.*)"
```

This example is actually where `WORKDIR` is excluded since `WORKDIR` is constructed as a path within `TMPDIR`, which is on the whitelist.

The `BB_HASHTASK_WHITELIST` covers dependent tasks and excludes certain kinds of tasks from the dependency chains. The effect of the previous example is to isolate the native, target, and cross-components. So, for example, toolchain changes do not force a rebuild of the whole system.

The end result of the "basic" handler is to make some dependency and hash information available to the build. This 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
```

There is also a "basichash" `BB_SIGNATURE_HANDLER`, which is the same as the basic 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. Currently, this behavior is not the default behavior. However, it is likely that the Yocto Project team will go forward with this behavior in the future since all the functionality exists. The reason for the delay is the potential impact to the distribution feed creation as they need increasing PR fields and the Yocto Project currently lacks a mechanism to automate incrementing this field.

4.2.3. 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 doesn't 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 mutliple 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/ \n \
file://.* file:///some/local/dir/sstate/"
```

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

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

4.2.4. Tips and Tricks

The code in the Yocto Project 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.

4.2.4.1. 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 BitBake is run 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 these `.siginfo` files. If one file is specified, it will dump out the dependency information in the file. If two files are specified, it will compare the two files and dump out the differences between the two. This allows the question of "What changed between X and Y?" to be answered easily.

4.2.4.2. Invalidating Shared State

The shared state code uses checksums and shared state memory 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. Let's say 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 would use 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 a change, you can take steps to invalidate the cache and force the task to run. The step to take is as simple as changing a function's comments in the source code. For example, to invalidate package shared state files, change the comment statments 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 [http://git.yoctoproject.org/cgit.cgi/poky/commit/meta/classes/package.bbclass?id=737f8bbb4f27b4837047cb9b4fbfe01dfde36d54].

Chapter 5. Board Support Packages (BSP) - Developer's Guide

A Board Support Package (BSP) is a collection of information that defines how to support a particular hardware device, set of devices, or hardware platform. The BSP includes information about the hardware features present on the device and kernel configuration information along with any additional hardware drivers required. The BSP also lists any additional software components required in addition to a generic Linux software stack for both essential and optional platform features.

This section (or document if you are reading the BSP Developer's Guide) defines a structure for these components so that BSPs follow a commonly understood layout. Providing a common form allows end-users to understand and become familiar with the layout. A common form also encourages standardization of software support of hardware.

> **Note**
>
> The information here does not provide an example of how to create a BSP. For examples on how to create a BSP, see the BSP Development Example [http://www.yoctoproject.org/docs/1.1.1/dev-manual/dev-manual.html#dev-manual-bsp-appendix] in The Yocto Project Development Manual [http://www.yoctoproject.org/docs/1.1.1/dev-manual/dev-manual.html]. You can also see the wiki page [https://wiki.yoctoproject.org/wiki/Transcript:_creating_one_generic_Atom_BSP_from_another].

The proposed format does have elements that are specific to the Yocto Project and OpenEmbedded build systems. It is intended that this information can be used by other systems besides Yocto Project and OpenEmbedded and that it will be simple to extract information and convert it to other formats if required. Yocto Project, through its standard layers mechanism, can directly accept the format described as a layer. The BSP captures all the hardware-specific details in one place in a standard format, which is useful for any person wishing to use the hardware platform regardless of the build system they are using.

The BSP specification does not include a build system or other tools - it is concerned with the hardware-specific components only. At the end distribution point you can ship the BSP combined with a build system and other tools. However, it is important to maintain the distinction that these are separate components that happen to be combined in certain end products.

5.1. Example Filesystem Layout

The BSP consists of a file structure inside a base directory, which uses the following naming convention:

```
meta-<bsp_name>
```

"bsp_name" is a placeholder for the machine or platform name. Here are some example base directory names:

```
meta-emenlow
meta-n450
meta-beagleboard
```

The base directory (`meta-<bsp_name>`) is the root of the BSP layer. This root is what you add to the BBLAYERS variable in the `build/conf/bblayers.conf` file found in the Yocto Project file's build directory. Adding the root allows the Yocto Project build system to recognize the BSP definition and from it build an image. Here is an example:

```
BBLAYERS = " \
   /usr/local/src/yocto/meta \
   /usr/local/src/yocto/meta-yocto \
```

```
        /usr/local/src/yocto/meta-<bsp_name> \
        "
```

For more detailed information on layers, see the "BitBake Layers [http://www.yoctoproject.org/docs/1.1.1/poky-ref-manual/poky-ref-manual.html#usingpoky-changes-layers]" section of the Yocto Project Reference Manual. You can also see the detailed examples in the appendices of The Yocto Project Development Manual [http://www.yoctoproject.org/docs/1.1.1/dev-manual/dev-manual.html].

Below is the common form for the file structure inside a base directory. While you can use this basic form for the standard, realize that the actual structures for specific BSPs could differ.

```
meta-<bsp_name>/
meta-<bsp_name>/<bsp_license_file>
meta-<bsp_name>/README
meta-<bsp_name>/binary/<bootable_images>
meta-<bsp_name>/conf/layer.conf
meta-<bsp_name>/conf/machine/*.conf
meta-<bsp_name>/recipes-bsp/*
meta-<bsp_name>/recipes-graphics/*
meta-<bsp_name>/recipes-kernel/linux/linux-yocto_<kernel_rev>.bbappend
```

Below is an example of the Crown Bay BSP:

```
meta-crownbay/COPYING.MIT
meta-crownbay/README
meta-crownbay/binary
meta-crownbay/conf/
meta-crownbay/conf/layer.conf
meta-crownbay/conf/machine/
meta-crownbay/conf/machine/crownbay.conf
meta-crownbay/conf/machine/crownbay-noemgd.conf
meta-crownbay/recipes-bsp/
meta-crownbay/recipes-bsp/formfactor/
meta-crownbay/recipes-bsp/formfactor/formfactor_0.0.bbappend
meta-crownbay/recipes-bsp/formfactor/formfactor/
meta-crownbay/recipes-bsp/formfactor/formfactor/crownbay/
meta-crownbay/recipes-bsp/formfactor/formfactor/crownbay/machconfig
meta-crownbay/recipes-bsp/formfactor/formfactor/crownbay-noemgd/
meta-crownbay/recipes-bsp/formfactor/formfactor/crownbay-noemgd/machconfig
meta-crownbay/recipes-core
meta-crownbay/recipes-core/tasks
meta-crownbay/recipes-core/tasks/task-core-tools.bbappend
meta-crownbay/recipes-graphics/
meta-crownbay/recipes-graphics/xorg-xserver/
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config_0.1.bbappend
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config/
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config/crownbay/
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config/crownbay/xorg.conf
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config/crownbay-noemgd/
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config/crownbay-noemgd/xorg.conf
meta-crownbay/recipes-kernel/
meta-crownbay/recipes-kernel/linux/
meta-crownbay/recipes-kernel/linux/linux-yocto_2.6.34.bbappend
meta-crownbay/recipes-kernel/linux/linux-yocto_2.6.37.bbappend
meta-crownbay/recipes-kernel/linux/linux-yocto_3.0.bbappend
```

The following sections describe each part of the proposed BSP format.

5.1.1. License Files

You can find these files in the Yocto Project file's directory structure at:

```
meta-<bsp_name>/<bsp_license_file>
```

These optional files satisfy licensing requirements for the BSP. The type or types of files here can vary depending on the licensing requirements. For example, in the Crown Bay BSP all licensing requirements are handled with the `COPYING.MIT` file.

Licensing files can be MIT, BSD, GPLv*, and so forth. These files are recommended for the BSP but are optional and totally up to the BSP developer.

5.1.2. README File

You can find these files in the Yocto Project file's directory structure at:

```
meta-<bsp_name>/README
```

This file provides information on how to boot the live images that are optionally included in the `/binary` directory. The `README` file also provides special information needed for building the image.

Technically speaking a `README` is optional but it is highly recommended that every BSP has one.

5.1.3. Pre-built User Binaries

You can find these files in the Yocto Project file's directory structure at:

```
meta-<bsp_name>/binary/<bootable_images>
```

This optional area contains useful pre-built kernels and user-space filesystem images appropriate to the target system. This directory typically contains graphical (e.g. sato) and minimal live images when the BSP tarball has been created and made available in the Yocto Project website. You can use these kernels and images to get a system running and quickly get started on development tasks.

The exact types of binaries present are highly hardware-dependent. However, a README file should be present in the BSP file structure that explains how to use the kernels and images with the target hardware. If pre-built binaries are present, source code to meet licensing requirements must also be provided in some form.

5.1.4. Layer Configuration File

You can find this file in the Yocto Project file's directory structure at:

```
meta-<bsp_name>/conf/layer.conf
```

The `conf/layer.conf` file identifies the file structure as a Yocto Project layer, identifies the contents of the layer, and contains information about how Yocto Project should use it. Generally, a standard boilerplate file such as the following works. In the following example you would replace "bsp" and "_bsp" with the actual name of the BSP (i.e. <bsp_name> from the example template).

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

# We have a recipes directory containing .bb and .bbappend files, add to BBFILES
BBFILES := "${BBFILES} ${LAYERDIR}/recipes/*/*.bb \
            ${LAYERDIR}/recipes/*/*.bbappend"

BBFILE_COLLECTIONS += "bsp"
BBFILE_PATTERN_bsp := "^${LAYERDIR}/"
```

```
BBFILE_PRIORITY_bsp = "5"
```

This file simply makes BitBake aware of the recipes and configuration directories. This file must exist so that the Yocto Project build system can recognize the BSP.

5.1.5. Hardware Configuration Options

You can find these files in the Yocto Project file's directory structure at:

```
meta-<bsp_name>/conf/machine/*.conf
```

The machine files bind together all the information contained elsewhere in the BSP into a format that the Yocto Project build system can understand. If the BSP supports multiple machines, multiple machine configuration files can be present. These filenames correspond to the values to which users have set the `MACHINE` variable.

These files define things such as the kernel package to use (`PREFERRED_PROVIDER` of virtual/kernel), the hardware drivers to include in different types of images, any special software components that are needed, any bootloader information, and also any special image format requirements.

At least one machine file is required for a BSP layer. However, you can supply more than one file. For example, in the Crown Bay BSP shown earlier in this section, the `conf/machine` directory contains two configuration files: `crownbay.conf` and `crownbay-noemgd.conf`. The `crownbay.conf` file is used for the Crown Bay BSP that supports the Intel® Embedded Media and Graphics Driver (Intel® EMGD), while the `crownbay-noemgd.conf` file is used for the Crown Bay BSP that does not support the Intel® EMGD.

This `crownbay.conf` file could also include a hardware "tuning" file that is commonly used to define the the package architecture and specify optimization flags, which are carefully chosen to give best performance on a given processor.

Tuning files are found in the `meta/conf/machine/include` directory. To use them, you simply include them in the machine configuration file. For example, the Crown Bay BSP `crownbay.conf` has the following statement:

```
include conf/machine/include/tune-atom.inc
```

5.1.6. Miscellaneous Recipe Files

You can find these files in the Yocto Project file's directory structure at:

```
meta-<bsp_name>/recipes-bsp/*
```

This optional directory contains miscellaneous recipe files for the BSP. Most notably would be the formfactor files. For example, in the Crown Bay BSP there is the `formfactor_0.0.bbappend` file, which is an append file used to augment the recipe that starts the build. Furthermore, there are machine-specific settings used during the build that are defined by the `machconfig` files. In the Crown Bay example, two `machconfig` files exist: one that supports the Intel EMGD and one that does not:

```
meta-crownbay/recipes-bsp/formfactor/formfactor/crownbay/machconfig
meta-crownbay/recipes-bsp/formfactor/formfactor/crownbay-noemgd/machconfig
meta-crownbay/recipes-bsp/formfactor/formfactor_0.0.bbappend
```

Note

If a BSP does not have a formfactor entry, defaults are established according to the configuration script.

5.1.7. Core Recipe Files

You can find these files in the Yocto Project file's directory structure at:

```
meta-<bsp_name>/recipes-core/*
```

This directory contains recipe files for the core. For example, in the Crown Bay BSP there is the `task-core-tools.bbappend` file, which is an append file used to recommend that the SystemTap package be included as a package when the image is built.

5.1.8. Display Support Files

You can find these files in the Yocto Project file's directory structure at:

```
meta-<bsp_name>/recipes-graphics/*
```

This optional directory contains recipes for the BSP if it has special requirements for graphics support. All files that are needed for the BSP to support a display are kept here. For example, the Crown Bay BSP contains the following files that support building a BSP that supports and does not support the Intel EMGD:

```
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config_0.1.bbappend
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config/crownbay/xorg.conf
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config/crownbay-noemgd/xorg.conf
```

5.1.9. Linux Kernel Configuration

You can find these files in the Yocto Project file's directory structure at:

```
meta-<bsp_name>/recipes-kernel/linux/linux-yocto_*.bbappend
```

These files append your specific changes to the kernel you are using.

For your BSP, you typically want to use an existing Yocto Project kernel found in the Yocto Project repository at `meta/recipes-kernel/linux`. You can append your specific changes to the kernel recipe by using a similarly named append file, which is located in the `meta-<bsp_name>/recipes-kernel/linux` directory.

Suppose you use a BSP that uses the `linux-yocto_3.0.bb` kernel, which is the preferred kernel to use for developing a new BSP using the Yocto Project. In other words, you have selected the kernel in your `<bsp_name>.conf` file by adding the following statements:

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
PREFERRED_VERSION_linux-yocto = "3.0%"
```

You would use the `linux-yocto_3.0.bbappend` file to append specific BSP settings to the kernel, thus configuring the kernel for your particular BSP.

As an example, look at the existing Crown Bay BSP. The append file used is:

```
meta-crownbay/recipes-kernel/linux/linux-yocto_3.0.bbappend
```

The file contains the following:

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

COMPATIBLE_MACHINE_crownbay = "crownbay"
KMACHINE_crownbay  = "yocto/standard/crownbay"
KERNEL_FEATURES_append_crownbay += " cfg/smp.scc"

COMPATIBLE_MACHINE_crownbay-noemgd = "crownbay-noemgd"
KMACHINE_crownbay-noemgd  = "yocto/standard/crownbay"
KERNEL_FEATURES_append_crownbay-noemgd += " cfg/smp.scc"

SRCREV_machine_pn-linux-yocto_crownbay ?= "2247da9131ea7e46ed4766a69bb1353dba22f873"
SRCREV_meta_pn-linux-yocto_crownbay ?= "d05450e4aef02c1b7137398ab3a9f8f96da74f52"

SRCREV_machine_pn-linux-yocto_crownbay-noemgd ?= "2247da9131ea7e46ed4766a69bb1353dba22f873"
SRCREV_meta_pn-linux-yocto_crownbay-noemgd ?= "d05450e4aef02c1b7137398ab3a9f8f96da74f52"
```

This append file contains statements used to support the Crown Bay BSP for both Intel EMGD and non-EMGD. The build process, in this case, recognizes and uses only the statements that apply to the defined machine name - `crownbay` in this case. So, the applicable statements in the `linux-yocto_3.0.bb`append file are follows:

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

COMPATIBLE_MACHINE_crownbay = "crownbay"
KMACHINE_crownbay  = "yocto/standard/crownbay"
KERNEL_FEATURES_append_crownbay += " cfg/smp.scc"

SRCREV_machine_pn-linux-yocto_crownbay ?= "2247da9131ea7e46ed4766a69bb1353dba22f873"
SRCREV_meta_pn-linux-yocto_crownbay ?= "d05450e4aef02c1b7137398ab3a9f8f96da74f52"
```

The append file defines `crownbay` as the compatible machine, defines the `KMACHINE`, points to some configuration fragments to use by setting the `KERNEL_FEATURES` variable, and then points to the specific commits in the Yocto Project files Git repository and the `meta` Git repository branches to identify the exact kernel needed to build the Crown Bay BSP.

One thing missing in this particular BSP, which you will typically need when developing a BSP, is the kernel configuration file (`.config`) for your BSP. When developing a BSP, you probably have a kernel configuration file or a set of kernel configuration files that, when taken together, define the kernel configuration for your BSP. You can accomplish this definition by putting the configurations in a file or a set of files inside a directory located at the same level as your append file and having the same name as the kernel. With all these conditions met simply reference those files in a `SRC_URI` statement in the append file.

For example, suppose you had a set of configuration options in a file called `defconfig`. If you put that file inside a directory named `/linux-yocto` and then added a `SRC_URI` statement such as the following to the append file, those configuration options will be picked up and applied when the kernel is built.

```
SRC_URI += "file://defconfig"
```

As mentioned earlier, you can group related configurations into multiple files and name them all in the `SRC_URI` statement as well. For example, you could group separate configurations specifically for Ethernet and graphics into their own files and add those by using a `SRC_URI` statement like the following in your append file:

```
SRC_URI += "file://defconfig \
       file://eth.cfg \
       file://gfx.cfg"
```

The FILESEXTRAPATHS variable is in boilerplate form here in order to make it easy to do that. It basically allows those configuration files to be found by the build process.

Note

Other methods exist to accomplish grouping and defining configuration options. For example, you could directly add configuration options to the Yocto kernel meta branch for your BSP. The configuration options will likely end up in that location anyway if the BSP gets added to the Yocto Project. For information on how to add these configurations directly, see The Yocto Project Kernel Architecture and Use Manual [http://yoctoproject.org/docs/1.1.1/kernel-manual/kernel-manual.html].

In general, however, the Yocto Project maintainers take care of moving the SRC_URI-specified configuration options to the meta branch. Not only is it easier for BSP developers to not have to worry about putting those configurations in the branch, but having the maintainers do it allows them to apply 'global' knowledge about the kinds of common configuration options multiple BSPs in the tree are typically using. This allows for promotion of common configurations into common features.

5.2. BSP 'Click-Through' Licensing Procedure

Note

This section describes how click-through licensing is expected to work. Currently, this functionality is not yet implemented.

In some cases, a BSP contains separately licensed IP (Intellectual Property) for a component that imposes upon the user a requirement to accept the terms of a 'click-through' license. Once the license is accepted the Yocto Project build system can then build and include the corresponding component in the final BSP image. Some affected components might be essential to the normal functioning of the system and have no 'free' replacement (i.e. the resulting system would be non-functional without them). On the other hand, other components might be simply 'good-to-have' or purely elective, or if essential nonetheless have a 'free' (possibly less-capable) version that could be used as a in the BSP recipe.

For cases where you can substitute something and still maintain functionality, the Yocto Project website's BSP Download Page [http://www.yoctoproject.org/download/all?keys=&download_type=1&download_version=] makes available 'de-featured' BSPs that are completely free of any IP encumbrances. For these cases you can use the substitution directly and without any further licensing requirements. If present, these fully 'de-featured' BSPs are named appropriately different as compared to the names of the respective encumbered BSPs. If available, these substitutions are the simplest and most preferred options. This, of course, assumes the resulting functionality meets requirements.

If however, a non-encumbered version is unavailable or the 'free' version would provide unsuitable functionality or quality, you can use an encumbered version.

Several methods exist within the Yocto Project build system to satisfy the licensing requirements for an encumbered BSP. The following list describes them in preferential order:

1. Get a license key (or keys) for the encumbered BSP by visiting a website and providing the name of the BSP and your email address through a web form.

   After agreeing to any applicable license terms, the BSP key(s) will be immediately sent to the address you gave and you can use them by specifying BSPKEY_<keydomain> environment variables when building the image:

   ```
   $ BSPKEY_<keydomain>=<key> bitbake core-image-sato
   ```

   These steps allow the encumbered image to be built with no change at all to the normal build process.

   Equivalently and probably more conveniently, a line for each key can instead be put into the user's local.conf file found in the Yocto Project file's build directory.

The <keydomain> component of the BSPKEY_<keydomain> is required because there might be multiple licenses in effect for a given BSP. In such cases, a given <keydomain> corresponds to a particular license. In order for an encumbered BSP that encompasses multiple key domains to be built successfully, a <keydomain> entry for each applicable license must be present in `local.conf` or supplied on the command-line.

2. Do nothing - build as you normally would. When a license is needed the build will stop and prompt you with instructions. Follow the license prompts that originate from the encumbered BSP. These prompts usually take the form of instructions needed to manually fetch the encumbered package(s) and md5 sums into the required directory (e.g. the `yocto/build/downloads`). Once the manual package fetch has been completed, restart the build to continue where it left off. During the build the prompt will not appear again since you have satisfied the requirement.

3. Get a full-featured BSP recipe rather than a key. You can do this by visiting the applicable BSP download page from the Yocto Project website at http://yoctoproject.org/download/board-support-package-bsp-downloads. BSP tarballs that have proprietary information can be downloaded after agreeing to licensing requirements as part of the download process. Obtaining the code this way allows you to build an encumbered image with no changes at all as compared to the normal build.

Note that the third method is also the only option available when downloading pre-compiled images generated from non-free BSPs. Those images are likewise available at from the Yocto Project website.

Chapter 6. Platform Development with the Yocto Project

6.1. Application Development Using the Yocto Project

The Yocto Project supports several methods of application development through which you can create user-space software designed to run on an embedded device that uses a Linux Yocto image developed with the Yocto Project. This flexibility allows you to choose the method that works best for you. This chapter describes each development method.

6.1.1. External Development Using the Meta-Toolchain

The Yocto Project provides toolchains that allow you to develop your application outside of the Yocto Project build system for specific hardware. These toolchains (called meta-toolchains) contain cross-development tools such as compilers, linkers, and debuggers that build your application for your target device. The Yocto Project also provides images that have toolchains for supported architectures included within the image. This allows you to compile, debug, or profile applications directly on the target device. See Appendix D,Reference: Images for a listing of the image types that Yocto Project supports.

Using the BitBake tool you can build a meta-toolchain or meta-toolchain-sdk target, which generates a tarball. Unpacking this tarball into the `/opt/poky` directory on your host produces a setup script (e.g. `/opt/poky/environment-setup-i586-poky-linux`) that you can `source` to initialize your build environment. Sourcing this script adds the compiler, QEMU scripts, QEMU binary, a special version of `pkgconfig` and other useful utilities to the PATH variable used by the Yocto Project build environment. Variables to assist `pkgconfig` and Autotools are also defined so that, for example, `configure` can find pre-generated test results for tests that need target hardware on which to run.

Using the toolchain with Autotool-enabled packages is straightforward - just pass the appropriate `host` option to `configure`. Following is an example:

```
$ ./configure --host=arm-poky-linux-gnueabi
```

For projects that are not Autotool-enabled, it is usually just a case of ensuring you point to and use the cross-toolchain. For example, the following two lines of code in a `Makefile` that builds your application specify to use the cross-compiler `arm-poky-linux-gnueabi-gcc` and linker `arm-poky-linux-gnueabi-ld`, which are part of the meta-toolchain you would have previously established:

```
CC=arm-poky-linux-gnueabi-gcc;
LD=arm-poky-linux-gnueabi-ld;
```

6.1.2. External Development Using the Eclipse Plug-in

The current release of the Yocto Project supports the Eclipse IDE plug-in to make developing software easier for the application developer. The plug-in provides capability extensions to the graphical IDE to allow for cross compilation, deployment and execution of the application within a QEMU emulation session. Support of the Eclipse plug-in also allows for cross debugging and profiling. Additionally, the Eclipse plug-in provides a suite of tools that allows the developer to perform remote profiling, tracing, collection of power consumption data, collection of latency data and collection of performance data.

> **Note**
>
> The current release of the Yocto Project no longer supports the Anjuta plug-in. However, the Poky Anjuta Plug-in is available to download directly from the Poky Git repository located through the web interface at http://git.yoctoproject.org/ under IDE Plugins. The community is free to continue supporting it beyond the Yocto Project 0.9 Release.

To use the Eclipse plug-in you need the Eclipse Framework (Helios 3.6.1) along with other plug-ins installed into the Eclipse IDE. Once you have your environment setup you need to configure the Eclipse plug-in. For information on how to install and configure the Eclipse plug-in, see the "Working Within Eclipse" [http://www.yoctoproject.org/docs/1.1.1/adt-manual/adt-manual.html#adt-eclipse] chapter in the "Application Development Toolkit (ADT) User's Guide." [http://www.yoctoproject.org/docs/1.1.1/adt-manual/adt-manual.html]

6.1.3. External Development Using the QEMU Emulator

Running Poky QEMU images is covered in the Yocto Project Quick Start [http://www.yoctoproject.org/docs/1.1.1/yocto-project-qs/yocto-project-qs.html] in the "A Quick Test Run" section.

The QEMU images shipped with the Yocto Project contain complete toolchains native to their target architectures. This support allows you to develop applications within QEMU similar to the way you would using a normal host development system.

Speed can be an issue depending on the target and host architecture mix. For example, using the `qemux86` image in the emulator on an Intel-based 32-bit (x86) host machine is fast because the target and host architectures match. On the other hand, using the `qemuarm` image on the same Intel-based host can be slower. But, you still achieve faithful emulation of ARM-specific issues.

To speed things up, the QEMU images support using `distcc` to call a cross-compiler outside the emulated system. If you used `runqemu` to start QEMU, and `distccd` is present on the host system, any BitBake cross-compiling toolchain available from the build system is automatically used from within QEMU simply by calling `distcc`. You can accomplish this by defining the cross-compiler variable (e.g. `export CC="distcc"`). Alternatively, if a suitable SDK/toolchain is present in `/opt/poky` the toolchain is also automatically used.

Several mechanisms exist that let you connect to the system running on the QEMU emulator:

- QEMU provides a framebuffer interface that makes standard consoles available.
- Generally, headless embedded devices have a serial port. If so, you can configure the operating system of the running image to use that port to run a console. The connection uses standard IP networking.
- The QEMU images have a Dropbear secure shell (ssh) server that runs with the root password disabled. This allows you to use standard `ssh` and `scp` commands.
- The QEMU images also contain an embedded Network Files System (NFS) server that exports the image's root filesystem. This allows you to make the filesystem available to the host.

6.1.4. Development Using Yocto Project Directly

Working directly with the Yocto Project is a fast and effective development technique. The idea is that you can directly edit files in a working directory (WORKDIR) or the source directory (S) and then force specific tasks to rerun in order to test the changes. An example session working on the matchbox-desktop package might look like this:

```
$ bitbake matchbox-desktop
$ sh
$ cd tmp/work/armv5te-poky-linux-gnueabi/matchbox-desktop-2.0+svnr1708-r0/
$ cd matchbox-desktop-2
$ vi src/main.c
      .
      .
 [Make your changes]
      .
      .
$ exit
$ bitbake matchbox-desktop -c compile -f
$ bitbake matchbox-desktop
```

This example builds the `matchbox-desktop` package, creates a new terminal, changes into the work directory for the package, changes a file, exits out of the terminal, and then recompiles the package.

Instead of using sh, you can also use two different terminals. However, the risk of using two terminals is that a command like unpack could destroy your changes in the work directory. Consequently, you need to work carefully.

It is useful when making changes directly to the work directory files to do so using the Quilt tool as detailed in the Modifying Package Source Code with Quilt section. Using Quilt, you can copy patches into the recipe directory and use the patches directly through use of the SRC_URI variable.

For a review of the skills used in this section, see the BitBake and Running Specific Tasks sections in this manual.

6.1.5. Development Within a Development Shell

When debugging certain commands or even when just editing packages, `devshell` can be a useful tool. Using `devshell` differs from the example shown in the previous section in that when you invoke `devshell` source files are extracted into your working directory and patches are applied. Then, a new terminal is opened and you are placed in the working directory. In the new terminal all the Yocto Project build-related environment variables are still defined so you can use commands such as `configure` and `make`. The commands execute just as if the Yocto Project build system were executing them. Consequently, workng this way can be helpful when debugging a build or preparing software to be used with the Yocto Project build system.

Following is an example that uses `devshell` on a target named `matchbox-desktop`:

```
$ bitbake matchbox-desktop -c devshell
```

This command opens a terminal with a shell prompt within the Poky environment. The following occurs:

- The PATH variable includes the cross-toolchain.
- The `pkgconfig` variables find the correct `.pc` files.
- The `configure` command finds the Yocto Project site files as well as any other necessary files.

Within this environment, you can run `configure` or `compile` commands as if they were being run by the Yocto Project build system itself. As noted earlier, the working directory also automatically changes to the source directory (S).

When you are finished, you just exit the shell or close the terminal window.

The default shell used by `devshell` is xterm. You can use other terminal forms by setting the TERMCMD and TERMCMDRUN variables in the Yocto Project's `local.conf` file found in the build directory. For examples of the other options available, see the "UI/Interaction Configuration" section of the `meta/conf/bitbake.conf` configuration file in the Yocto Project files.

Because an external shell is launched rather than opening directly into the original terminal window, it allows easier interaction with Bitbake's multiple threads as well as accomodates a future client/server split.

Note

It is worth remembering that when using `devshell` you need to use the full compiler name such as `arm-poky-linux-gnueabi-gcc` instead of just using `gcc`. The same applies to other applications such as `binutils`, `libtool` and so forth. The Yocto Project has setup environment variables such as CC to assist applications, such as `make` to find the correct tools.

It is also worth noting that `devshell` still works over X11 forwarding and similar situations

6.1.6. Development Within Yocto Project for a Package that Uses an External SCM

If you're working on a recipe that pulls from an external Source Code Manager (SCM), it is possible to have the Yocto Project build system notice new changes added to the SCM and then build the package

that depends on them using the latest version. This only works for SCMs from which it is possible to get a sensible revision number for changes. Currently, you can do this with Apache Subversion (SVN), Git, and Bazaar (BZR) repositories.

To enable this behavior, simply add the following to the `local.conf` configuration file in the build directory of the Yocto Project files:

```
SRCREV_pn-<PN> = "${AUTOREV}"
```

where PN is the name of the package for which you want to enable automatic source revision updating.

6.2. Debugging With the GNU Project Debugger (GDB) Remotely

GDB allows you to examine running programs, which in turn help you to understand and fix problems. It also allows you to perform post-mortem style analysis of program crashes. GDB is available as a package within the Yocto Project and by default is installed in sdk images. See Appendix D,Reference: Images for a description of these images. You can find information on GDB at http://sourceware.org/gdb/.

Tip

For best results, install `-dbg` packages for the applications you are going to debug. Doing so makes available extra debug symbols that give you more meaningful output.

Sometimes, due to memory or disk space constraints, it is not possible to use GDB directly on the remote target to debug applications. These constraints arise because GDB needs to load the debugging information and the binaries of the process being debugged. Additionally, GDB needs to perform many computations to locate information such as function names, variable names and values, stack traces and so forth - even before starting the debugging process. These extra computations place more load on the target system and can alter the characteristics of the program being debugged.

To help get past the previously mentioned constraints, you can use Gdbserver. Gdbserver runs on the remote target and does not load any debugging information from the debugged process. Instead, a GDB instance processes the debugging information that is run on a remote computer - the host GDB. The host GDB then sends control commands to Gdbserver to make it stop or start the debugged program, as well as read or write memory regions of that debugged program. All the debugging information loaded and processed as well as all the heavy debugging is done by the host GDB. Offloading these processes gives the Gdbserver running on the target a chance to remain small and fast.

Because the host GDB is responsible for loading the debugging information and for doing the necessary processing to make actual debugging happen, the user has to make sure the host can access the unstripped binaries complete with their debugging information and also be sure the target is compiled with no optimizations. The host GDB must also have local access to all the libraries used by the debugged program. Because Gdbserver does not need any local debugging information, the binaries on the remote target can remain stripped. However, the binaries must also be compiled without optimization so they match the host's binaries.

To remain consistent with GDB documentation and terminology, the binary being debugged on the remote target machine is referred to as the "inferior" binary. For documentation on GDB see the GDB site [http://sourceware.org/gdb/documentation/].

6.2.1. Launching Gdbserver on the Target

First, make sure Gdbserver is installed on the target. If it is not, install the package `gdbserver`, which needs the `libthread-db1` package.

As an example, to launch Gdbserver on the target and make it ready to "debug" a program located at `/path/to/inferior`, connect to the target and launch:

```
$ gdbserver localhost:2345 /path/to/inferior
```

Gdbserver should now be listening on port 2345 for debugging commands coming from a remote GDB process that is running on the host computer. Communication between Gdbserver and the host GDB are done using TCP. To use other communication protocols, please refer to the Gdbserver documentation [http://www.gnu.org/software/gdb/].

6.2.2. Launching GDB on the Host Computer

Running GDB on the host computer takes a number of stages. This section describes those stages.

6.2.2.1. Building the Cross-GDB Package

A suitable GDB cross-binary is required that runs on your host computer but also knows about the the ABI of the remote target. You can get this binary from the the Yocto Project meta-toolchain. Here is an example:

```
/usr/local/poky/eabi-glibc/arm/bin/arm-poky-linux-gnueabi-gdb
```

where `arm` is the target architecture and `linux-gnueabi` the target ABI.

Alternatively, the Yocto Project can build the `gdb-cross` binary. Here is an example:

```
$ bitbake gdb-cross
```

Once the binary is built, you can find it here:

```
tmp/sysroots/<host-arch>/usr/bin/<target-abi>-gdb
```

6.2.2.2. Making the Inferior Binaries Available

The inferior binary (complete with all debugging symbols) as well as any libraries (and their debugging symbols) on which the inferior binary depends need to be available. There are a number of ways you can make these available.

Perhaps the easiest way is to have an 'sdk' image that corresponds to the plain image installed on the device. In the case of `core-image-sato`, `core-image-sdk` would contain suitable symbols. Because the sdk images already have the debugging symbols installed, it is just a question of expanding the archive to some location and then informing GDB.

Alternatively, Yocto Project can build a custom directory of files for a specific debugging purpose by reusing its `tmp/rootfs` directory. This directory contains the contents of the last built image. This process assumes two things:

- The image running on the target was the last image to be built by the Yocto Project.
- The package (`foo` in the following example) that contains the inferior binary to be debugged has been built without optimization and has debugging information available.

The following steps show how to build the custom directory of files:

1. Install the package (`foo` in this case) to `tmp/rootfs`:

```
$ tmp/sysroots/i686-linux/usr/bin/opkg-cl -f \
tmp/work/<target-abi>/core-image-sato-1.0-r0/temp/opkg.conf -o \
tmp/rootfs/ update
```

2. Install the debugging information:

```
$ tmp/sysroots/i686-linux/usr/bin/opkg-cl -f \
tmp/work/<target-abi>/core-image-sato-1.0-r0/temp/opkg.conf \
-o tmp/rootfs install foo

$ tmp/sysroots/i686-linux/usr/bin/opkg-cl -f \
tmp/work/<target-abi>/core-image-sato-1.0-r0/temp/opkg.conf \
-o tmp/rootfs install foo-dbg
```

6.2.2.3. Launch the Host GDB

To launch the host GDB, you run the `cross-gdb` binary and provide the inferior binary as part of the command line. For example, the following command form continues with the example used in the previous section. This command form loads the `foo` binary as well as the debugging information:

```
$ <target-abi>-gdb rootfs/usr/bin/foo
```

Once the GDB prompt appears, you must instruct GDB to load all the libraries of the inferior binary from `tmp/rootfs` as follows:

```
$ set solib-absolute-prefix /path/to/tmp/rootfs
```

The pathname `/path/to/tmp/rootfs` must either be the absolute path to `tmp/rootfs` or the location at which binaries with debugging information reside.

At this point you can have GDB connect to the Gdbserver that is running on the remote target by using the following command form:

```
$ target remote remote-target-ip-address:2345
```

The `remote-target-ip-address` is the IP address of the remote target where the Gdbserver is running. Port 2345 is the port on which the GDBSERVER is running.

6.2.2.4. Using the Debugger

You can now proceed with debugging as normal - as if you were debugging on the local machine. For example, to instruct GDB to break in the "main" function and then continue with execution of the inferior binary use the following commands from within GDB:

```
(gdb) break main
(gdb) continue
```

For more information about using GDB, see the project's online documentation at http://sourceware.org/gdb/download/onlinedocs/.

6.3. Profiling with OProfile

OProfile [http://oprofile.sourceforge.net/] is a statistical profiler well suited for finding performance bottlenecks in both userspace software and in the kernel. This profiler provides answers to questions like "Which functions does my application spend the most time in when doing X?" Because the Yocto Project is well integrated with OProfile, it makes profiling applications on target hardware straightforward.

To use OProfile, you need an image that has OProfile installed. The easiest way to do this is with `tools-profile` in the `IMAGE_FEATURES` variable. You also need debugging symbols to be available on the system where the analysis takes place. You can gain access to the symbols by using `dbg-pkgs` in the `IMAGE_FEATURES` variable or by installing the appropriate `-dbg` packages.

For successful call graph analysis, the binaries must preserve the frame pointer register and should also be compiled with the `-fno-omit-framepointer` flag. In the Yocto Project you can achieve this by setting the `SELECTED_OPTIMIZATION` variable to `-fexpensive-optimizations -fno-omit-framepointer -frename-registers -O2`. You can also achieve it by setting the `DEBUG_BUILD` variable to "1" in the `local.conf` configuration file. If you use the `DEBUG_BUILD` variable you will also add extra debug information that can make the debug packages large.

6.3.1. Profiling on the Target

Using OProfile you can perform all the profiling work on the target device. A simple OProfile session might look like the following:

```
# opcontrol --reset
# opcontrol --start --separate=lib --no-vmlinux -c 5
         .
         .
   [do whatever is being profiled]
         .
         .
# opcontrol --stop
$ opreport -cl
```

In this example, the `reset` command clears any previously profiled data. The next command starts OProfile. The options used when starting the profiler separate dynamic library data within applications, disable kernel profiling, and enable callgraphing up to five levels deep.

Note

To profile the kernel, you would specify the `--vmlinux=/path/to/vmlinux` option. The `vmlinux` file is usually in the Yocto Project file's `/boot/` directory and must match the running kernel.

After you perform your profiling tasks, the next command stops the profiler. After that, you can view results with the `opreport` command with options to see the separate library symbols and callgraph information.

Callgraphing logs information about time spent in functions and about a function's calling function (parent) and called functions (children). The higher the callgraphing depth, the more accurate the results. However, higher depths also increase the logging overhead. Consequently, you should take care when setting the callgraphing depth.

Note

On ARM, binaries need to have the frame pointer enabled for callgraphing to work. To accomplish this use the `-fno-omit-framepointer` option with `gcc`.

For more information on using OProfile, see the OProfile online documentation at http://oprofile.sourceforge.net/docs/.

6.3.2. Using OProfileUI

A graphical user interface for OProfile is also available. You can download and build this interface from the Yocto Project at http://git.yoctoproject.org/cgit.cgi/oprofileui/. If the "tools-profile" image feature is selected, all necessary binaries are installed onto the target device for OProfileUI interaction.

Even though the Yocto Project usually includes all needed patches on the target device, you might find you need other OProfile patches for recent OProfileUI features. If so, see the OProfileUI README [http://git.yoctoproject.org/cgit.cgi/oprofileui/tree/README] for the most recent information.

6.3.2.1. Online Mode

Using OProfile in online mode assumes a working network connection with the target hardware. With this connection, you just need to run "oprofile-server" on the device. By default, OProfile listens on port 4224.

Note

You can change the port using the `--port` command-line option.

The client program is called `oprofile-viewer` and its UI is relatively straightforward. You access key functionality through the buttons on the toolbar, which are duplicated in the menus. Here are the buttons:

- Connect: Connects to the remote host. You can also supply the IP address or hostname.
- Disconnect: Disconnects from the target.
- Start: Starts profiling on the device.
- Stop: Stops profiling on the device and downloads the data to the local host. Stopping the profiler generates the profile and displays it in the viewer.
- Download: Downloads the data from the target and generates the profile, which appears in the viewer.
- Reset: Resets the sample data on the device. Resetting the data removes sample information collected from previous sampling runs. Be sure you reset the data if you do not want to include old sample information.
- Save: Saves the data downloaded from the target to another directory for later examination.
- Open: Loads previously saved data.

The client downloads the complete 'profile archive' from the target to the host for processing. This archive is a directory that contains the sample data, the object files, and the debug information for the object files. The archive is then converted using the `oparchconv` script, which is included in this distribution. The script uses `opimport` to convert the archive from the target to something that can be processed on the host.

Downloaded archives reside in the Yocto Project's build directory in `/tmp` and are cleared up when they are no longer in use.

If you wish to perform kernel profiling, you need to be sure a `vmlinux` file that matches the running kernel is available. In the Yocto Project, that file is usually located in `/boot/vmlinux-KERNELVERSION`, where `KERNEL-version` is the version of the kernel. The Yocto Project generates separate `vmlinux` packages for each kernel it builds. Thus, it should just be a question of making sure a matching package is installed (e.g. `opkg install kernel-vmlinux`. The files are automatically installed into development and profiling images alongside OProfile. A configuration option exists within the OProfileUI settings page that you can use to enter the location of the `vmlinux` file.

Waiting for debug symbols to transfer from the device can be slow, and it is not always necessary to actually have them on the device for OProfile use. All that is needed is a copy of the filesystem with the debug symbols present on the viewer system. The Launching GDB on the Host Computer section covers how to create such a directory with the Yocto Project and how to use the OProfileUI Settings dialog to specify the location. If you specify the directory, it will be used when the file checksums match those on the system you are profiling.

6.3.2.2. Offline Mode

If network access to the target is unavailable, you can generate an archive for processing in `oprofile-viewer` as follows:

```
# opcontrol --reset
# opcontrol --start --separate=lib --no-vmlinux -c 5
       .
       .
[do whatever is being profiled]
       .
       .
# opcontrol --stop
# oparchive -o my_archive
```

In the above example, `my_archive` is the name of the archive directory where you would like the profile archive to be kept. After the directory is created, you can copy it to another host and load it using `oprofile-viewer` open functionality. If necessary, the archive is converted.

Appendix A. Reference: Directory Structure

The Yocto Project consists of several components. Understanding them and knowing where they are located is key to using the Yocto Project well. This appendix describes the Yocto Project file's directory structure and gives information about the various files and directories.

For information on how to establish the Yocto Project files on your local development system, see the Getting Setup [http://www.yoctoproject.org/docs/1.1.1/dev-manual/dev-manual.html#getting-started] section in the The Yocto Project Development Manual [http://www.yoctoproject.org/docs/1.1.1/dev-manual/dev-manual.html].

A.1. Top level core components

A.1.1. **bitbake/**

The Yocto Project 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. The `bitbake/bin/` directory is placed into the PATH environment variable by the oe-init-build-env script.

For more information on BitBake, see the BitBake on-line manual at http://bitbake.berlios.de/manual/.

A.1.2. **build/**

This directory contains user configuration files and the output generated by the Yocto Project in its standard configuration where the source tree is combined with the output. The build directory is created initially when you `source` the Yocto Project environment setup script `oe-init-build-env`.

It is also possible to place output and configuration files in a directory separate from the Yocto Project files by providing a directory name when you `source` the setup script. For information on separating output from the Yocto Project files, see oe-init-build-env.

A.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 `poky-ref-manual`.

A.1.4. **meta/**

This directory contains the Yocto Project core metadata. The directory holds machine definitions, the Yocto Project distribution, and the packages that make up a given system.

A.1.5. **meta-demoapps/**

This directory contains recipes for applications and demos that are not part of the Yocto Project core.

A.1.6. **meta-rt/**

This directory contains recipes for real-time kernels.

A.1.7. **meta-skeleton/**

This directory contains template recipes for BSP and kernel development.

A.1.8. **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 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`.

A.1.9. **oe-init-build-env**

This script sets up the Yocto Project 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 the Yocto Project to create a build directory of your choice. For example, the following command creates a build directory named `mybuilds` that is outside of the Yocto Project files:

```
$ source oe-init-build-env ~/mybuilds
```

A.1.10. **LICENSE, README, and README.hardware**

These files are standard top-level files.

A.2. The Build Directory -**build/**

A.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.

A.2.2. **build/conf/local.conf**

This file contains all the local user configuration of the Yocto Project. 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 Yocto Project unless that variable is hard-coded within the Yocto Project (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`), or where you want to downloaded files (`DL_DIR`).

A.2.3. **build/conf/bblayers.conf**

This file defines layers, which is a directory tree, 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.

A.2.4. **build/conf/sanity_info**

This file is created during the build to indicate the state of the sanity checks.

A.2.5. **build/downloads/**

This directory is used for the upstream source tarballs. The directory can be reused by multiple builds or moved to another location. You can control the location of this directory through the `DL_DIR` variable.

A.2.6. **build/sstate-cache/**

This directory is used for the shared state cache. The directory can be reused by multiple builds or moved to another location. You can control the location of this directory through the `SSTATE_DIR` variable.

A.2.7. **build/tmp/**

This directory receives all the Yocto Project output. BitBake creates this directory if it does not exist. As a last resort, to clean the Yocto Project and start a build from scratch (other than downloads), you can remove everything in this directory or get rid of the directory completely. If you do, you should also completely remove the `build/sstate-cache` directory as well.

A.2.8. **build/tmp/buildstats/**

This directory stores the build statistics.

A.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. These results are stored here on a per-machine basis.

A.2.10. **build/tmp/deploy/**

This directory contains any 'end result' output from the Yocto Project build process.

A.2.11. **build/tmp/deploy/deb/**

This directory receives any `.deb` packages produced by the Yocto Project. The packages are sorted into feeds for different architecture types.

A.2.12. **build/tmp/deploy/rpm/**

This directory receives any `.rpm` packages produced by the Yocto Project. The packages are sorted into feeds for different architecture types.

A.2.13. **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.

Note, you should not remove any files from this directory by hand in an attempt to rebuild an image. If you want to clean out the cache, re-run the build using the following BitBake command:

```
$ bitbake -c cleanall <target>
```

A.2.14. **build/tmp/deploy/ipk/**

This directory receives `.ipk` packages produced by the Yocto Project.

A.2.15. **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.

A.2.16. **build/tmp/stamps/**

This directory holds information that that BitBake uses for accounting purposes to track what tasks have run and when they have run. The directory is sub-divided by architecture. The files in the

directory are empty of data. However, BitBake uses the filenames and timestamps for tracking purposes.

A.2.17. **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.

A.2.18. **build/tmp/pkgdata/**

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

A.2.19. **build/tmp/work/**

This directory contains architecture-specific work sub-directories for packages built by BitBake. All tasks execute from a 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 for which the source is being compiled.

It is worth considering the structure of a typical work directory. As an example, consider the 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 WORKDIR, is created. Within this directory, the source is unpacked to linux-qemux86-standard-build and then patched by Quilt (see the Modifying Package Source Code With Quilt section). 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/.

A.3. The Metadata -**meta/**

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

A.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.

A.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 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.

A.3.3. **meta/conf/machine/**

This directory contains all the machine configuration files. If you set MACHINE="qemux86", Yocto Project 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.

A.3.4. **meta/conf/distro/**

Any distribution-specific configuration is controlled from this directory. The Yocto Project only contains the Yocto Project distribution so `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 is `poky-bleeding.conf` although this file mainly inherits its configuration from the Yocto Project itself.

A.3.5. **meta/recipes-bsp/**

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

A.3.6. **meta/recipes-connectivity/**

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

A.3.7. **meta/recipes-core/**

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

A.3.8. **meta/recipes-devtools/**

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

A.3.9. **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.

A.3.10. **meta/recipes-gnome/**

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

A.3.11. **meta/recipes-graphics/**

This directory contains X and other graphically related system libraries

A.3.12. **meta/recipes-kernel/**

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

A.3.13. **meta/recipes-multimedia/**

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

A.3.14. **meta/recipes-qt/**

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

A.3.15. **meta/recipes-sato/**

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

A.3.16. **meta/recipes-support/**

This directory contains recipes that used by other recipes, but that are not directly included in images (i.e. dependencies of other recipes).

A.3.17. **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.

A.3.18. **meta/recipes.txt/**

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

Appendix B. Reference: BitBake

BitBake is a program written in Python that interprets the metadata that makes up the Yocto Project. At some point, developers wonder what actually happens when you enter:

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

This appendix 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.

B.1. Parsing

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

The first thing BitBake does is look for the `bitbake.conf` file. The Yocto Project keeps this file in the Yocto Project file's `meta/conf/` directory. BitBake finds it by examining the `BBPATH` environment variable and looking for the `meta/conf/` directory.

In the Yocto Project, `bitbake.conf` 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 Yocto Project 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` 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 BitBake Layers section.

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 or class files the `.bb` file depends on changes.

B.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 `task-base.bb`, which in turn leads to packages like `Contacts`, `Dates` and `BusyBox`. These packages 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.

B.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 manual [http://bitbake.berlios.de/manual/]. At a basic level, it is sufficient to know that BitBake uses the DEPENDS and RDEPENDS variables when calculating dependencies.

B.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 [http://www.yoctoproject.org/docs/1.1.1/yocto-project-qs/yocto-project-qs.html#building-image] section in the Yocto Project Quick Start [http://www.yoctoproject.org/docs/1.1.1/yocto-project-qs/yocto-project-qs.html] for more information.

As each task completes, a timestamp is written to the directory specified by the STAMPS variable (usually `build/tmp/stamps/*/`). On subsequent runs, BitBake looks at the 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.

B.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 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_WHITELIST` 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 Section 4.2.2, “Checksums (Signatures)”.

B.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
```

B.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 [http://bitbake.berlios.de/manual/].

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 Development Within Yocto Project for a Package that Uses an External SCM section for more information.

Appendix C. Reference: Classes

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 Yocto Project file's area 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 behaviour.

C.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`.

C.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 as documented in the Autotooled Package section and then simply `inherit autotools`. This class can also work with software that emulates Autotools.

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 DESTDIR option, which takes its value from the standard DESTDIR variable.

C.3. Alternatives -**update-alternatives.bbclass**

Several programs can fulfill the same or similar function and be 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.

Four variables control this class:

- `ALTERNATIVE_NAME` – The name of the binary that is replaced (`ar` in this example).
- `ALTERNATIVE_LINK` – The path to the resulting binary (`/bin/ar` in this example).
- `ALTERNATIVE_PATH` – The path to the real binary (`/usr/bin/ar.binutils` in this example).
- `ALTERNATIVE_PRIORITY` – The priority of the binary. The version with the most features should have the highest priority.

Currently, the Yocto Project supports only one binary per package.

C.4. Initscripts -**update-rc.d.bbclass**

This class uses `update-rc.d` to safely install an initialization script on behalf of the package. The Yocto Project 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.

C.5. Binary config 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.

C.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`.

C.7. Pkg-config -**pkgconfig.bbclass**

`pkg-config` brought standardization and this class aims to make its integration smooth for all libraries that make use of 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.

C.8. Distribution of sources - **src_distribute_local.bbclass**

Many software licenses require that source files be provided along with the binaries. To simplify this process, two classes were created: `src_distribute.bbclass` and `src_distribute_local.bbclass`.

The results of these classes are `tmp/deploy/source/` subdirs with sources sorted by `LICENSE` field. If recipes list few licenses (or have entries like "Bitstream Vera"), the source archive is placed in each license directory.

This class operates using three modes:

- copy: Copies the files to the distribute directory.
- symlink: Symlinks the files to the distribute directory.
- move+symlink: Moves the files into the distribute directory and then symlinks them back.

C.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.

C.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 dependling 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.

C.11. Developer Shell -**devshell.bbclass**

This class adds the `devshell` task. Distribution policy dictates whether to include this class as the Yocto Project does. See the Development Within a Development Shell section for more information about using devshell.

C.12. 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 found in the Yocto Project file's `conf` 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.

The package class you choose can affect build-time performance and has space ramifications. In general, building a package with RPM takes about thirty percent more time as compared to using IPK 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.

Keep in mind, however, that 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.

Another consideration for packages built using the RPM package manager is space. For smaller systems, the extra space used for the Berkley Database and the amount of metadata can affect your ability to do 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/006362.html]
- https://lists.yoctoproject.org/pipermail/poky/2011-May/006363.html [https://lists.yoctoproject.org/pipermail/poky/2011-May/006363.html]

C.13. 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.

C.14. 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.

C.15. Host System sanity checks - **sanity.bbclass**

This class checks to see if prerequisite software is present 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 whether to include this class as the Yocto Project does.

C.16. 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 Yocto Project. 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 as the Yocto Project does.

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 it matches 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.
- `la`: Checks `.la` files for any `TMPDIR` paths. Any `.la` file continaing 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.

C.17. 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.

C.18. 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 Yocto Project Files provides a simple exmample 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.

C.19. Other Classes

Thus far, this appendix has discussed only the most useful and important classes. However, other classes exist within the `meta/classes` directory in the Yocto Project file's directory structure. You can examine the `.bbclass` files directly for more information.

Appendix D. Reference: Images

The Yocto Project 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 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`, and `meta/recipes-sato/images` directories of your local Yocto Project file structure (Git repository or extracted release tarball). Although the recipe names are somewhat explanatory, here is a list that describes them:

- `core-image-base`: A console-only image that fully supports the target device hardware.
- `core-image-core`: An X11 image with simple applications such as terminal, editor, and file manager.
- `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.
- `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 foundational basic image without support for X that can be reasonably used for customization.
- `core-image-lsb`: A `core-image-basic` image suitable for implementations that conform to Linux Standard Base (LSB).
- `core-image-lsb-dev`: A `core-image-lsb` image that is suitable for development work.
- `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. See the External Development Using the Poky SDK section for more information.
- `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-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 Pimlico applications and also contains terminal, editor, and file manager.
- `core-image-sato-dev`: A `core-image-sato` image suitable for development that also includes a native toolchain and libraries needed to build applications on the device itself. The image also

includes testing and profiling tools as well as 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. See the External Development Using the Poky SDK section for more information.

  ## 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.

Appendix E. Reference: Features

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.

E.1. Distro

The items below are valid options for DISTRO_FEATURES:

- alsa: ALSA support will be included (OSS compatibility kernel modules will be installed if available).
- bluetooth: Include bluetooth support (integrated BT only)
- ext2: Include tools for supporting for devices with internal HDD/Microdrive for storing files (instead of Flash only devices)
- irda: Include Irda support
- keyboard: Include keyboard support (e.g. keymaps will be loaded during boot).
- pci: Include PCI bus support
- pcmcia: Include PCMCIA/CompactFlash support
- usbgadget: USB Gadget Device support (for USB networking/serial/storage)
- usbhost: USB Host support (allows to connect external keyboard, mouse, storage, network etc)
- wifi: WiFi support (integrated only)
- cramfs: CramFS support
- ipsec: IPSec support
- ipv6: IPv6 support
- nfs: NFS client support (for mounting NFS exports on device)
- ppp: PPP dialup support
- smbfs: SMB networks client support (for mounting Samba/Microsoft Windows shares on device)

E.2. Machine

The items below are valid options for MACHINE_FEATURES:

- 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

E.3. Reference: Images

The contents of images generated by the Yocto Project 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:

- apps-console-core: Core console applications such as `ssh`, `daemon`, `avahi daemon`, `portmap` (for mounting NFS shares)
- x11-base: X11 server + minimal desktop
- x11-sato: OpenedHand Sato environment
- apps-x11-core: Core X11 applications such as an X Terminal, file manager, and file editor
- apps-x11-games: A set of X11 games
- apps-x11-pimlico: OpenedHand Pimlico application suite
- tools-sdk: A full SDK that runs on the device
- tools-debug: Debugging tools such as `strace` and `gdb`
- tools-profile: Profiling tools such as `oprofile`, `exmap`, and LTTng
- tools-testapps: Device testing tools (e.g. touchscreen debugging)
- nfs-server: NFS server (exports / over NFS to everybody)
- dev-pkgs: Development packages (headers and extra library links) for all packages installed in a given image
- dbg-pkgs: Debug packages for all packages installed in a given image

Appendix F. Reference: Variables Glossary

This section lists common variables used in the Yocto Project and gives an overview of their function and contents.

Glossary

A B C D E F H I K L M P R S T W

A

AUTHOR — The email address used to contact the original author or authors in order to send patches, forward bugs, etc.

AUTOREV — Specifies to use the current (newest) source revision. This variable is with the SRCREV variable.

B

BAD_RECOMMENDATIONS — A list of packages not to install despite being recommended by a recipe. Support for this variable exists only for images that use the `ipkg` packaging system.

BB_NUMBER_THREADS — The maximum number of tasks BitBake should run in parallel at any one time. If your host development system supports mulitiple 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 package appears in more than one layer. Setting this variable allows you to prioritize a layer against other layers that contain the same package - effectively letting you control the precedence for the multiple layers. The precedence established through this variable stands regardless of a layer's package version (PV variable). For example, a layer that has a package 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 Yocto Project build. This variable is defined in the `bblayers.conf` configuration file in the Yocto Project build directory. Here is an example:

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

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

BPN — Bare name of package with any suffixes like -cross -native removed.

C

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

COMPATIBLE_MACHINE — A regular expression which evaluates to match the machines the recipe works with. It stops recipes being run on machines for which they are not compatible. This is particularly useful with kernels. It also helps to increase parsing speed as further parsing of the recipe is skipped if it is found the current machine is not compatible.

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`.

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 the priority of recipes.

DEPENDS — A list of build-time dependencies for a given recipe. The variable indicates recipes that must have been staged before a particular recipe can configure.

DESCRIPTION — The package description used by package managers.

DESTDIR — the destination directory.

DISTRO — The short name of the distribution.

DISTRO_EXTRA_RDEPENDS — The list of packages required by the distribution.

DISTRO_EXTRA_RRECOMMENDS

The list of packages which extend usability of the image. Those packages will automatically be installed but can be removed by user.

DISTRO_FEATURES — The features of the distribution.

DISTRO_NAME — The long name of the distribution.

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

See Handling a Package Name Alias section for more information.

DISTRO_VERSION — the version of the distribution.

DL_DIR — The directory where all fetched sources will be stored.

E

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

EXTRA_IMAGE_FEATURES — Allows extra packages to be added to the generated images. You set this variable in the `local.conf` configuration file. Note that some image features are also added using the `IMAGE_FEATURES` variable generally configured in image recipes. You can use this variable to add more features in addition to those. Here are some examples of features you can add:

```
"dbg-pkgs" - Adds -dbg packages for all installed packages
             including symbol information for debugging and
             profiling.

"dev-pkgs" - Adds -dev packages for all installed packages.
             This is useful if you want to develop against
             the libraries in the image.

"tools-sdk" - Adds development tools such as gcc, make,
              pkgconfig and so forth.

"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-testapps" - Adds useful testing tools such as
                   ts_print, aplay, arecord and so
                   forth.

"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.

                 There are other application targets too, see
                 meta/classes/poky-image.bbclass
                 and meta/packages/tasks/task-poky.bb
                 for more details.
```

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.

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 Yocto Project uses the `fs-perms.txt`, which is located in the `meta/files` directory of the Yocto Project files 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 Yocto Project's 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 optimised system. This variable defaults to "-fexpensive-optimizations -fomit-frame-pointer -frename-registers -O2".

H

HOMEPAGE — Website where more info about package can be found

I

IMAGE_FEATURES — The list of features present in images. Typically, you configure this variable in image recipes. Note that you can add extra features to the image by using the `EXTRA_IMAGE_FEATURES` variable. See the Reference: Images section for the list of features present in images built by the Yocto Project.

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

IMAGE_INSTALL — The list of packages used to build images.

INC_PR — Defines the Package revision. You manually combine values for `INC_PR` into the PR field of the parent recipe. When you change this variable, you change the PR value for every person that includes the file.

The following example shows how to use the `INC_PR` variable given a common `.inc` file that defines the variable. Once defined, you can use the variable to set the PR value:

```
recipes-graphics/xorg-font/font-util_1.1.1.bb:PR - "$(INC_PR).1"
recipes-graphics/xorg-font/xorg-font-common.inc:INC_PR - "r1"
recipes-graphics/xorg-font/encondings_1.0.3.bb:PR - "$(INC_PR).1"
recipes-graphics/xorg-font/fiont-alias_1.0.2.bb:PR - "$(INC_PR).0"
```

INHIBIT_PACKAGE_STRIP — 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`. An example is `start 99 5 2 . stop 20 0 1 6 .`, which gives the script 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`.

K

KERNEL_FEATURES

Includes additional metadata from the Linux Yocto kernel Git repository. In the Yocto Project 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 Linux Yocto 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.

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 requires immediate expansion (see the BitBake manual) as lazy expansion can result in the expansion happening in the wrong directory and therefore giving the wrong value.

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`).

LICENSE
The list of package source licenses.

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 Track License Change section

M

MACHINE
Specifies the target device.

MACHINE_ESSENTIAL_EXTRA_RDEPENDS
A list of required packages to install as part of the package 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 `task-core-boot`, including the `core-image-minimal` image.

This variable is similar to the MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS variable with the exception that the package 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.

For example, suppose you are building a runtime package that depends on a certain disk driver. In this case, you would use the following:

```
MACHINE_ESSENTIAL_EXTRA_RDEPENDS += "<disk_driver>"
```

MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS
A list of recommended packages to install as part of the package being built. The build process does not depend 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 `task-core-boot`, including the `core-image-minimal` image.

This variable is similar to the MACHINE_ESSENTIAL_EXTRA_RDEPENDS variable with the exception that the package 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. However, because this is one of the "essential" variables, the resulting image might not boot on the machine. Or, if

the machine does boot using the image, the machine might not be fully functional.

Consider an example where you have a custom kernel with a disk driver built into the kernel itself, rather than using the driver built as a module. If you include the package that has the driver module as part of the variable's list, the build process will not find that package. However, because these packages are "recommends" packages, the build will not fail due to the missing package. Not accounting for any other problems, the custom kernel would still boot the machine.

Some example packages of these machine essentials are flash, screen, keyboard, mouse, or touchscreen drivers (depending on the machine).

For example, suppose you are building a runtime package that depends on a mouse driver. In this case, you would use the following:

```
MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS += "<mouse_driver>"
```

MACHINE_EXTRA_RDEPENDS

A list of optional but non-machine essential packages to install as part of the package being built. Even though these packages are not essential for the machine to boot, the build process depends on them being present. The impact of this variable affects all images based on `task-base`, which does not include the `core-image-minimal` or `core-image-basic` images.

This variable is similar to the MACHINE_EXTRA_RRECOMMENDS variable with the exception that the package 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 might or might not have a WiFi card. The package containing the WiFi support is not essential for the machine to boot the image. If it is not there, the machine will boot but not be able to use the WiFi functionality. However, if you include the package with the WiFi support as part of the variable's package list, the build process depends on finding the package. In this case, you would use the following:

```
MACHINE_EXTRA_RDEPENDS += "<wifi_driver>"
```

MACHINE_EXTRA_RRECOMMENDS

A list of optional but non-machine essential packages to install as part of the package being built. The package being built has no build dependency on the list of packages with this variable. The impact of this variable affects only images based on `task-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 package 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 might or might not have a WiFi card. The package containing the WiFi support is not essential for the machine to boot the image. If it is not there, the machine will boot but not be able to use the WiFi functionality. You are free to either include or not include the the package with the WiFi support as part of the variable's package list, the build process does not depend on finding the package. If you include the package, you would use the following:

```
MACHINE_EXTRA_RRECOMMENDS += "<wifi_driver>"
```

MACHINE_FEATURES — Specifies the list of device features. See the Machine section for more information.

MAINTAINER — The email address of the distribution maintainer.

P

PACKAGE_ARCH — The architecture of the resulting package.

PACKAGE_CLASSES — This variable, which is set in the `local.conf` configuration file found in the Yocto Project file's `conf` 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_ipk"
```

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

PACKAGE_DESCRIPTION — The long form description of the binary package for packaging systems such as `ipkg`, `rpm` or `debian`. By default, this variable inherits `DESCRIPTION`.

PACKAGE_EXTRA_ARCHS — Specifies the list of architectures compatible with the device CPU. This variable is useful when you build for several different devices that use miscellaneous processors such as XScale and ARM926-EJS).

PACKAGE_SUMMARY — The short (72 character limit suggested) summary of the binary package for packaging systems such as `ipkg`, `rpm` or `debian`. By default, this variable inherits `DESCRIPTION`.

PACKAGES — The list of packages to be created from the recipe. The default value is "`${PN}`-dbg `${PN}` `${PN}`-doc `${PN}`-dev".

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 one and a half times the number of cores on the host.

PN — The name of the package.

PR — The revision of the package. The default value for this variable is "r0".

PV — The version of the package. The version is normally extracted from the recipe name. For example, if the recipe is named `expat_2.0.1.bb`, then `PV` will be `2.0.1`. `PV` is generally not overridden within a recipe unless it is building an unstable version from a source code repository (e.g. Git or Subversion).

PE — the epoch of the package. 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. The variable must always be suffixed with the name of the provided item, and should be set 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. The variable must always be suffixed with the $PN for which to select, and should be set to the $PV to which you want to give 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%"
```

POKY_EXTRA_INSTALL

Specifies the list of packages to be added to the image. This variable should only be set in the `local.conf` configuration file found in the Yocto Project's build directory.

POKYLIBC

This variable is no longer supported and has been replaced by the TCLIBC variable.

POKYMODE

This variable is no longer supported and has been replaced by the TCMODE variable.

R

RCONFLICTS

The list of packages that conflict with this package. Note that the package will not be installed if the conflicting packages are not first removed.

RDEPENDS

A list of packages that must be installed as part of a package being built. The package being built has a runtime dependency on the packages in the variable's list. In other words, in order for the package being built to run correctly, it depends on these listed packages. If a package in this list cannot be found during the build, the build will not complete.

Because the RDEPENDS variable applies to packages being built, you should always attach an override to the variable to specify the particular runtime package that has the dependency. 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 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`.

Some automatic handling occurs around the RDEPENDS variable:

- `shlibdeps`: If a runtime 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 to create the runtime 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 runtime packages.

ROOT_FLASH_SIZE — The size of root filesystem as measured in megabytes.

RRECOMMENDS — A list of packages that extend 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 Yocto Project 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 — The list of packages that are replaced with this package.

S

S — The path to unpacked sources. By default, this path is "${WORKDIR}/${PN}-${PV}".

SECTION — The section where package should be put. Package managers use 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.

SSTATE_DIR — The directory for the shared state.

SHELLCMDS — A list of commands to run within the shell. The list is used by TERMCMDRUN.

SITEINFO_ENDIANNESS — Specifies the endian byte order of the target system. The variable is either "le" for little-endian or "be" for big-endian.

SITEINFO_BITS — Specifies the number of bits for the target system CPU. The variable is either "32" or "64".

SRC_URI — The list of source files - local or remote.

SRC_URI_OVERRIDES_PACKAGE_ARCH

By default, the Yocto Project automatically detects whether SRC_URI contains files that are machine-specific. If so, the Yocto Project automatically changes PACKAGE_ARCH. Setting this variable to "0" disables this behaviour.

SRCDATE — The date of the source code used to build the package. This variable applies only if the source was fetched from a Source Code Manager (SCM).

SRCREV — The revision of the source code used to build the package. This variable applies 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.

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

STAMPS — The directory (usually TMPDIR/stamps) with timestamps of executed tasks.

SUMMARY — The short (72 characters or less) summary of the binary package for packaging systems such as ipkg, rpm or debian. By default, this variable inherits DESCRIPTION.

T

TARGET_ARCH — The architecture of the device being built. While a number of values are possible, the Yocto Project primarily supports arm and i586.

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 from the Yocto Project or a few supported combinations of the upstream GCC or CodeSourcery Labs toolchain. The variable determines which of the files in meta/conf/distro/include/tcmode-* is used.

By default, TCMODE is set to "default", which chooses tcmode-default.inc. 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`.

TERMCMD

This command is used by BitBake to launch a terminal window with a shell. The shell is unspecified so the user's default shell is used. By default, the variable is set to "xterm" but it can be any X11 terminal application or a terminal multiplexer such as screen.

> **Note**
>
> While `KONSOLE_TERMCMD` and `KONSOLE_TERMCMDRUN` are provided and will work with KDE's Konsole terminal application Konsole from KDE 3, Konsole in KDE 4.0 and later versions will no longer work here due to the fact that it now launches in the background by default, and it is not practically possible to wait until it has terminated. It is hoped that this can be fixed in a future version.

TERMCMDRUN

This variable is similar to `TERMCMD`. However, instead of running the user's shell, the command specified by the `SHELLCMDS` variable is run.

W

WORKDIR

The path to directory in `tmp/work/` where the package is built.

Appendix G. Reference: Variable Context

While most variables can be used in almost any context such as `.conf`, `.bbclass`, `.inc`, and `.bb` files, some variables are often associated with a particular locality or context. This appendix describes some common associations.

G.1. Configuration

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

G.1.1. Distribution (Distro)

This section lists variables whose context is the distribution, or distro.

- `DISTRO`
- `DISTRO_NAME`
- `DISTRO_VERSION`
- `MAINTAINER`
- `PACKAGE_CLASSES`
- `TARGET_OS`
- `TARGET_FPU`
- `POKYMODE`
- `TCMODE`
- `POKYLIBC`

G.1.2. Machine

This section lists variables whose context is the machine.

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

G.1.3. Local

This section lists variables whose 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

G.2. Recipes

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

G.2.1. Required

This section lists variables that are required for recipes.

- DESCRIPTION
- LICENSE
- LIC_FILES_CHKSUM
- SECTION
- HOMEPAGE
- AUTHOR
- SRC_URI

G.2.2. Dependencies

This section lists variables that define recipe dependencies.

- DEPENDS
- RDEPENDS
- RRECOMMENDS
- RCONFLICTS
- RREPLACES

G.2.3. Paths

This section lists variables that define recipe paths.

- WORKDIR
- S
- FILES

G.2.4. Extra Build Information

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

- DISTRO_PN_ALIAS
- EXTRA_OECMAKE
- EXTRA_OECONF
- EXTRA_OEMAKE
- PACKAGES
- DEFAULT_PREFERENCE

Appendix H. FAQ

H.1. How does Poky differ from OpenEmbedded [http://www.openembedded.org/]?

Poky is the Yocto Project build system that was derived from OpenEmbedded [http://www.openembedded.org/]. Poky is a stable, smaller subset focused on the mobile environment. Development in the Yocto Project using Poky is closely tied to OpenEmbedded with features being merged regularly between the two for mutual benefit.

H.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 [http://downloads.yoctoproject.org/releases/miscsupport/yocto-1.0-python-nativesdk/python-nativesdk-standalone-i686.tar.bz2]
- 64-bit tarball [http://downloads.yoctoproject.org/releases/miscsupport/yocto-1.0-python-nativesdk/python-nativesdk-standalone-x86_64.tar.bz2]

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.

H.3. How can you claim Poky is stable?

There are three areas that help with stability;

- The Yocto Project team keeps Poky small and focused. It contains around 650 packages as compared to over 5000 for full OpenEmbedded.
- The Yocto Project only supports hardware that the team has access to for testing.
- The Yocto Project uses an an autobuilder, which provides continuous build and integration tests.

H.4. How do I get support for my board added to the Yocto Project?

There are two main ways to get a board supported in the Yocto Project;

- Send the Yocto Project team information on the board and if the team does not have it yet they will consider adding it.
- Send the Yocto Project team the BitBake recipes if you have them.

Usually, if the board is not completely exotic, adding support in the Yocto Project is fairly straightforward.

H.5. Are there any products using Poky?

The Vernier LabQuest [http://vernier.com/labquest/] is using the Yocto Project build system Poky. See the Vernier LabQuest [http://www.vernier.com/products/interfaces/labq/] for more information. There are a number of pre-production devices using Poky and the Yocto Project team announces them as soon as they are released.

H.6. What does the Yocto Project build system Poky produce as output?

Because the same set of recipes can be used to create output of various formats, the output of a Yocto Project build depends on how it was started. Usually, the output is a flashable image ready for the target device.

H.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 Adding a Package section earlier in this manual.

H.8. Do I have to reflash my entire board with a new Yocto Project image when recompiling a package?

The Yocto Project can build packages in various formats such as `ipk` for `ipkg`/`opkg`, Debian package (`.deb`), or RPM. The packages can then be upgraded using the package tools on the device, much like on a desktop distribution such as Ubuntu or Fedora.

H.9. What is GNOME Mobile and what is the difference between GNOME Mobile and GNOME?

GNOME Mobile [http://www.gnome.org/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.

H.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.

H.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 [http://dag.wieers.com/rpm/packages/python-sqlite2/]
- `help2man` from Karan repository [http://centos.karan.org/el5/extras/testing/i386/RPMS/help2man-1.33.1-2.noarch.rpm]

Once these packages are installed, the Yocto Project 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.

H.12. I see lots of 404 responses for files on `http://www.yoctoproject.org/sources/*`. Is something wrong?

Nothing is wrong. The Yocto Project checks any configured source mirrors before downloading from the upstream sources. The Yocto Project 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 Yocto Project. Consequently, if an upstream source disappears, the team can place sources there so builds continue to work.

H.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 in the case that needs it. The code that handles `SRC_URI_OVERRIDES_PACKAGE_ARCH` is in `base.bbclass`.

H.14. I'm behind a firewall and need to use a proxy server. How do I do that?

Most source fetching by the Yocto Project is done by `wget` and you therefore need to specify the proxy settings in a `.wgetrc` file in your home directory. Example settings in that file would be

```
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.

H.15. I'm using Ubuntu Intrepid and am seeing build failures. What's wrong?

In Intrepid, Ubuntu turns on by default the normally optional compile-time security features and warnings. There are more details at https://wiki.ubuntu.com/CompilerFlags. You can work around this problem by disabling those options by adding the following to the `BUILD_CPPFLAGS` variable in the `conf/bitbake.conf` file.

```
" -Wno-format-security -U_FORTIFY_SOURCE"
```

H.16. 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.

H.17. I'm seeing random build failures. Help?!

If the same build is failing in totally different and random ways, the most likely explanation is that either the hardware you're running the build on has some problem, or, if you are running the build under virtualisation, the virtualisation probably has bugs. The Yocto Project processes a massive amount of data causing 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 virtualisation issues.

H.18. 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 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.

H.19. How do I disable the cursor on my touchscreen device?

You need to create a form factor file as described in Section 5.1.6, "Miscellaneous Recipe Files" and set the `HAVE_TOUCHSCREEN` variable equal to one as follows:

```
HAVE_TOUCHSCREEN=1
```

H.20. 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 Section 5.1.6, "Miscellaneous Recipe Files"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_4.44.bbappend
```

H.21. How do I create images with more free space?

Images are created to be 1.2 times the size of the populated root filesystem. To modify this ratio so that there is more free space available, you need to set the configuration value IMAGE_OVERHEAD_FACTOR. For example, setting IMAGE_OVERHEAD_FACTOR to 1.5 sets the image size ratio to one and a half times the size of the populated root filesystem.

```
IMAGE_OVERHEAD_FACTOR = "1.5"
```

H.22. 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 Yocto Project depends on such as autoconf break when they find spaces in pathnames. Until that situation changes, the team will not support spaces in pathnames.

H.23. 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 file to include (conf/distro/include/tcmode-*.inc).

The default value of TCMODE is "default". However, other patterns are accepted. In particular, "external-*" refers to external toolchains of which there are some basic examples included with the core. A user can use their own custom toolchain definition in their 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-csl-toolchain_2008q3-72.bb, which reuses the core libc packaging class to do most of the work.

H.24. How does the Yocto Project obtain source code and will it work behind my firewall or proxy server?

The way the Yocto Project obtains source code is highly configurable. You can setup the Yocto Project 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.

By default, Poky uses the Yocto Project source PREMIRRORS 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 Poky 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 Poky 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 throw 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 Poky 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 Poky 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 PREMIRROR server is up to date:

```
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 Poky to successfully fetch source over HTTP and any network accesses to anything other than the PREMIRROR would fail.

Poky also honors the standard environment variables `http_proxy`, `ftp_proxy`, `https_proxy`, and `all_proxy` to redirect requests through proxy servers.

Appendix I. Contributing to the Yocto Project

I.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 [http://www.yoctoproject.org/docs/1.1.1/dev-manual/dev-manual.html#local-yp-release] list item in The Yocto Project Development Manual [http://www.yoctoproject.org/docs/1.1.1/dev-manual/dev-manual.html].

I.2. Tracking Bugs

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

I.3. Mailing lists

To subscribe to the Yocto Project mailing lists, click on the following URLs and follow the instructions:

- http://lists.yoctoproject.org/listinfo/yocto-announce: Use this list to receive offical Yocto Project announcements for developments and to learn about Yocto Project milestones.
- http://lists.yoctoproject.org/listinfo/yocto: Use this list to monitor Yocto Project development discussions, ask questions, and get help.
- http://lists.yoctoproject.org/listinfo/poky: Use this list to monitor discussions about the Yocto Project build system Poky, ask questions, and get help.

I.4. Internet Relay Chat (IRC)

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

- `#yocto`
- `#poky`

I.5. Links

Following is a list of resources you will find helpful:

- The Yocto Project website [http://yoctoproject.org]: The home site for the Yocto Project.
- OpenedHand [http://www.openedhand.com/]: The company where the Yocto Project build system Poky was first developed. OpenedHand has since been acquired by Intel Corporation.
- Intel Corporation [http://www.intel.com/]: The company who acquired OpenedHand in 2008 and continues development on the Yocto Project.
- OpenEmbedded [http://www.openembedded.org/]: The upstream, generic, embedded distribution the Yocto Project build system (Poky) derives from and to which it contributes.
- Bitbake [http://developer.berlios.de/projects/bitbake/]: The tool used to process Yocto Project metadata.
- BitBake User Manual [http://bitbake.berlios.de/manual/]: A comprehensive guide to the BitBake tool.
- Pimlico [http://pimlico-project.org/]: A suite of lightweight Personal Information Management (PIM) applications designed primarily for handheld and mobile devices.

- QEMU [http://wiki.qemu.org/Index.html]: An open source machine emulator and virtualizer.

I.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, see How to Submit a Change [http://www.yoctoproject.org/docs/1.1.1/dev-manual/dev-manual.html#how-to-submit-a-change] in The Yocto Project Development Manual [http://www.yoctoproject.org/docs/1.1.1/dev-manual/dev-manual.html].