Poky Reference Manual

A Guide and Reference to Poky

Richard Purdie, Linux Foundation
<richard.purdie@linuxfoundation.org>
Tomas Frydrych, Intel Corporation
Marcin Juszkiewicz
Dodji Seketeli

Poky Reference Manual: A Guide and Reference to Poky

by Richard Purdie, Tomas Frydrych, Marcin Juszkiewicz, and Dodji Seketeli

Table of Contents

1. Introduction 1
    1.1. Welcome to Poky! 1
    1.2. What is Poky? 1
    1.3. Documentation Overview 2
    1.4. System Requirements 2
    1.5. Obtaining Poky 3
        1.5.1. Releases 3
        1.5.2. Nightly Builds 3
        1.5.3. Development Checkouts 3
2. Using Poky 4
    2.1. Poky Overview 4
        2.1.1. BitBake 4
        2.1.2. Metadata (Recipes) 4
        2.1.3. Classes 4
        2.1.4. Configuration 5
    2.2. Running a Build 5
    2.3. Installing and Using the Result 5
    2.4. Debugging Build Failures 5
        2.4.1. Task Failures 5
        2.4.2. Running Specific Tasks 5
        2.4.3. Dependency Graphs 6
        2.4.4. General BitBake Problems 6
        2.4.5. Building with No Dependencies 6
        2.4.6. Variables 6
        2.4.7. Other Tips 7
3. Extending Poky 8
    3.1. Adding a Package 8
        3.1.1. Single .c File Package (Hello World!) 8
        3.1.2. Autotooled Package 8
        3.1.3. Makefile-Based Package 9
        3.1.4. Controlling Package Content 9
        3.1.5. Post Install Scripts 10
    3.2. Customizing Images 10
        3.2.1. Customizing Images Using Custom .bb Files 10
        3.2.2. Customizing Images Using Custom Tasks 11
        3.2.3. Customizing Images Using Custom IMAGE_FEATURES 11
        3.2.4. Customizing Images Using local.conf 12
    3.3. Porting Poky to a New Machine 12
        3.3.1. Adding the Machine Configuration File 12
        3.3.2. Adding a Kernel for the Machine 12
        3.3.3. Adding a Formfactor Configuration File 13
    3.4. Making and Maintaining Changes 13
        3.4.1. BitBake Layers 13
        3.4.2. Committing Changes 14
        3.4.3. Package Revision Incrementing 15
        3.4.4. Using Poky in a Team Environment 15
        3.4.5. Updating Existing Images 16
    3.5. Modifying Package Source Code 16
        3.5.1. Modifying Package Source Code with quilt 16
    3.6. Track License Change 17
        3.6.1. Specifying the LIC_FILES_CHKSUM Variable 17
        3.6.2. Explanation of Syntax 17
    3.7. Handling Package Name Alias 18
        3.7.1. Specifying the DISTRO_PN_ALIAS Variable 18
4. Board Support Packages (BSP) - Developers Guide 19
    4.1. Example Filesystem Layout 19
        4.1.1. License Files 20
        4.1.2. README File 20
        4.1.3. Pre-built User Binaries 20
        4.1.4. Layer Configuration File 20
        4.1.5. Hardware Configuration Options 21

4.1.6. Miscellaneous Recipe Files 21
4.1.7. Display Support Files 22
4.1.8. Linux Kernel Configuration 22
4.2. BSP 'Click-Through' Licensing Procedure 23
5. Platform Development with Poky 25
5.1. Software development 25
5.1.1. External Development Using the Poky SDK 25
5.1.2. Using the Eclipse and Anjuta Plug-ins 25
5.1.3. Developing Externally in QEMU 29
5.1.4. Developing in Poky Directly 30
5.1.5. Developing with 'devshell' 30
5.1.6. Developing within Poky with an External SCM-based Package 31
5.2. Debugging with GDB Remotely 31
5.2.1. Launching GDBSERVER on the Target 31
5.2.2. Launching GDB on the Host Computer 32
5.3. Profiling with OProfile 33
5.3.1. Profiling on the Target 34
5.3.2. Using OProfileUI 34
A. Reference: Directory Structure 36
A.1. Top level core components 36
A.1.1. bitbake/ 36
A.1.2. build/ 36
A.1.3. meta/ 36
A.1.4. meta-extras/ 36
A.1.5. meta-***/ 36
A.1.6. scripts/ 36
A.1.7. sources/ 36
A.1.8. documentation 37
A.1.9. poky-init-build-env 37
A.2. The Build Directory - build/ 37
A.2.1. build/conf/local.conf 37
A.2.2. build/conf/bblayers.conf 37
A.2.3. build/tmp/ 37
A.2.4. build/tmp/cache/ 37
A.2.5. build/tmp/deploy/ 37
A.2.6. build/tmp/deploy/deb/ 38
A.2.7. build/tmp/deploy/rpm/ 38
A.2.8. build/tmp/deploy/images/ 38
A.2.9. build/tmp/deploy/ipk/ 38
A.2.10. build/tmp/sysroots/ 38
A.2.11. build/tmp/stamps/ 38
A.2.12. build/tmp/log/ 38
A.2.13. build/tmp/pkgdata/ 38
A.2.14. build/tmp/pstagelogs/ 38
A.2.15. build/tmp/work/ 38
A.3. The Metadata - meta/ 39
A.3.1. meta/classes/ 39
A.3.2. meta/conf/ 39
A.3.3. meta/conf/machine/ 39
A.3.4. meta/conf/distro/ 39
A.3.5. meta/recipes-bsp/ 39
A.3.6. meta/recipes-connectivity/ 39
A.3.7. meta/recipes-core/ 39
A.3.8. meta/recipes-devtools/ 39
A.3.9. meta/recipes-extended/ 39
A.3.10. meta/recipes-gnome/ 40
A.3.11. meta/recipes-graphics/ 40
A.3.12. meta/recipes-kernel/ 40
A.3.13. meta/recipes-multimedia/ 40
A.3.14. meta/recipes-qt/ 40
A.3.15. meta/recipes-sato/ 40
A.3.16. meta/site/ 40
B. Reference: BitBake 41
B.1. Parsing 41

B.2. Preferences and Providers 41
B.3. Dependencies 42
B.4. The Task List 42
B.5. Running a Task 42
B.6. BitBake Command Line 43
B.7. Fetchers 43
C. Reference: Classes 45
C.1. The base class - base.bbclass 45
C.2. Autotooled Packages - autotools.bbclass 45
C.3. Alternatives - update-alternatives.bbclass 45
C.4. Initscripts - update-rc.d.bbclass 46
C.5. Binary config scripts - binconfig.bbclass 46
C.6. Debian renaming - debian.bbclass 46
C.7. Pkg-config - pkgconfig.bbclass 46
C.8. Distribution of sources - src_distribute_local.bbclass 46
C.9. Perl modules - cpan.bbclass 46
C.10. Python extensions - distutils.bbclass 46
C.11. Developer Shell - devshell.bbclass 47
C.12. Packaging - package*.bbclass 47
C.13. Building kernels - kernel.bbclass 47
C.14. Creating images - image.bbclass and rootfs*.bbclass 47
C.15. Host System sanity checks - sanity.bbclass 47
C.16. Generated output quality assurance checks - insane.bbclass 47
C.17. Autotools configuration data cache - siteinfo.bbclass 48
C.18. Other Classes 48
D. Reference: Images 49
E. Reference: Features 50
E.1. Distro 50
E.2. Machine 50
E.3. Reference: Images 51
F. Reference: Variables Glossary 52
G. Reference: Variable Locality (Distro, Machine, Recipe etc.) 58
G.1. Distro Configuration 58
G.2. Machine Configuration 58
G.3. Local Configuration (local.conf) 58
G.4. Recipe Variables - Required 59
G.5. Recipe Variables - Dependencies 59
G.6. Recipe Variables - Paths 59
G.7. Recipe Variables - Extra Build Information 59
H. FAQ 60
I. Contributing to Poky 64
I.1. Introduction 64
I.2. Bugtracker 64
I.3. Mailing lists 64
I.4. Internet Relay Chat (IRC) 64
I.5. Links 64
I.6. Contributions 64

Chapter 1. Introduction

1.1. Welcome to Poky!

Poky is the build tool in the Yocto Project. The Yocto Project uses Poky to build images (kernel, system, and application software) for targeted hardware.

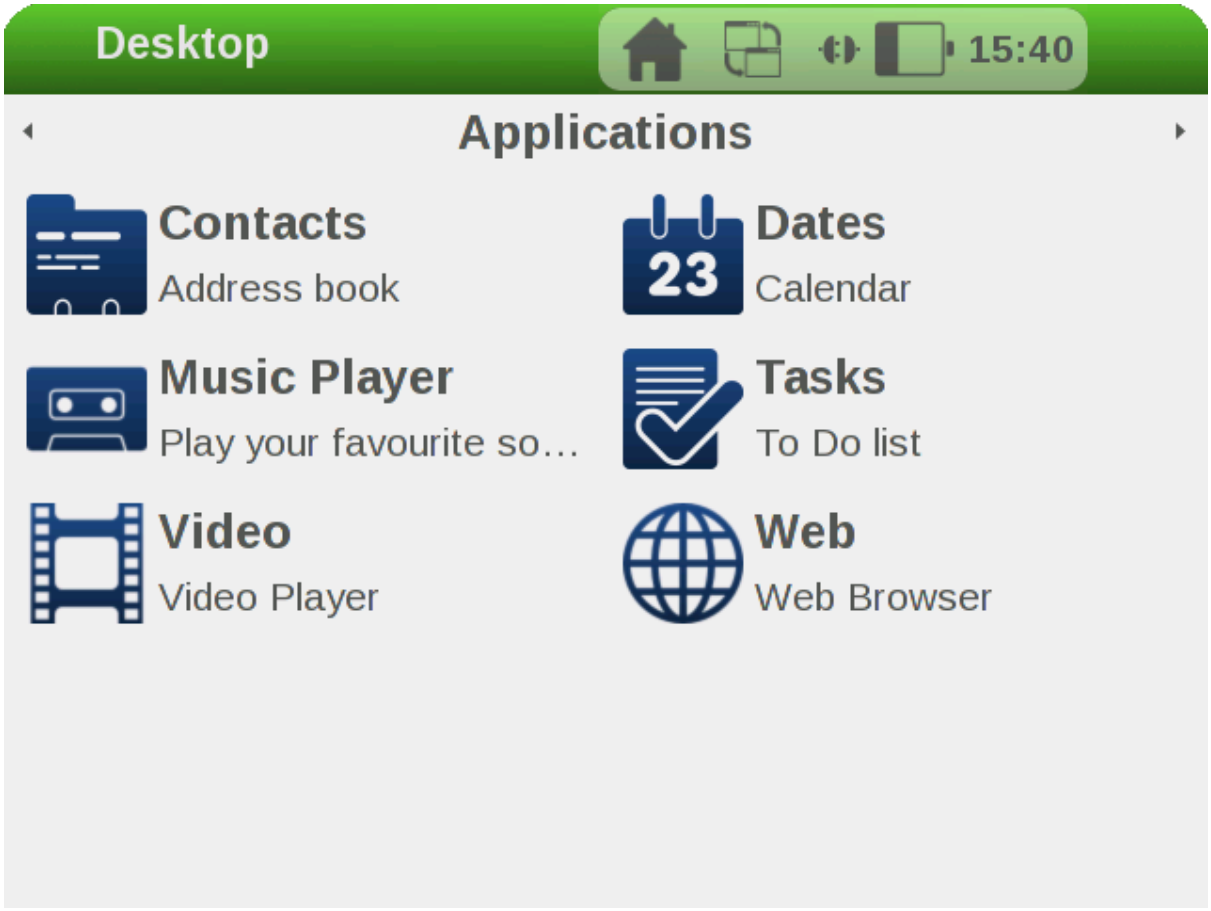
Before diving into Poky, it helps to have an understanding of the Yocto Project. Especially useful for newcomers is the information in the Yocto Project Quick Start, which you can find on the Yocto Project website [http://www.yoctoproject.org]. Specifically, the guide is at http://www.yoctoproject.org/docs/yocto-quick-start/yocto-project-qs.html.

1.2. What is Poky?

Within the Yocto Project, Poky provides an open source, full-platform build tool based on Linux, X11, Matchbox, GTK+, Pimlico, Clutter, and other GNOME Mobile [http://gnome.org/mobile] technologies. It provides a focused and stable subset of OpenEmbedded upon which you can easily and reliably build and develop. Poky fully supports a wide range of x86, ARM, MIPS and PowerPC hardware and device virtualization.

Poky is primarily a platform builder that generates filesystem images based on open source software such as the Kdrive X server, the Matchbox window manager, the GTK+ toolkit and the D-Bus message bus system. While images for many kinds of devices can be generated, the standard example machines target QEMU full-system emulation (x86, ARM, MIPS and PowerPC) and real reference boards for each of these architectures. Poky's ability to boot inside a QEMU emulator makes it particularly suitable as a test platform for developing embedded software.

An important component integrated within Poky is Sato, a GNOME Mobile-based user interface environment. It is designed to work well with screens that use very high DPI and have restricted sizes, such as those often found on smartphones and PDAs. Because Sato is coded for speed and efficiency, it works smoothly on hand-held and other embedded hardware. It sits nicely on top of any device that uses the GNOME Mobile stack and it results in a well-defined user experience.



The Sato Desktop - A screenshot from a machine running a Poky built image

Poky has a growing open source community and is also backed up by commercial organizations including Intel® Corporation.

1.3. Documentation Overview

The sections in this reference manual describe different aspects of Poky. The 'Using Poky' section provides an overview of the components that make up Poky followed by information about using Poky and debugging images created in the Yocto Project. The 'Extending Poky' and 'Board Support Packages' sections provide information about how to extend and customize Poky along with advice on how to manage these changes. The 'Platform Development with Poky' section provides information about interaction between Poky and target hardware for common platform development tasks such as software development, debugging and profiling. The rest of the manual consists of several reference sections, each providing details on a specific area of Poky functionality.

This manual applies to Poky Release 5.0 (Bernard).

1.4. System Requirements

Although we recommend Debian-based distributions (Ubuntu 10.04 or newer) as the host system for Poky, nothing in Poky is distribution-specific. Consequently, other distributions should work as long as the appropriate prerequisites are installed. For example, we know of Poky being used successfully on Redhat, SUSE, Gentoo and Slackware host systems. For information on what you need to develop images using Yocto Project and Poky, you should see the Yocto Project Quick Start on the Yocto Project website [http://www.yoctoproject.org]. The direct link to the quick start is http://yoctoproject.org/docs/yocto-quick-start/yocto-project-qs.html.

1.5. Obtaining Poky

1.5.1. Releases

Periodically, we make releases of Poky available at http://yoctoproject.org/downloads/poky/. These releases are more stable and more rigorously tested than the nightly development images.

1.5.2. Nightly Builds

We make nightly builds of Poky for testing purposes and to make the latest developments available. The output from these builds is available at http://autobuilder.yoctoproject.org/. The numbers used in the builds increase for each subsequent build and can be used to reference a specific build.

Automated builds are available for "standard" Poky and for Poky SDKs and toolchains. Additionally, testing versions such as poky-bleeding can be made available as 'experimental' builds. The toolchains can be used either as external standalone toolchains or can be combined with Poky as a pre-built toolchain to reduce build time. Using the external toolchains is simply a case of untarring the tarball into the root of your system (it only creates files in `/opt/poky`) and then enabling the option in `local.conf`.

1.5.3. Development Checkouts

Poky is available from our git repository located at git://git.yoctoproject.org/poky.git; a web interface to the repository can be accessed at http://git.yoctoproject.org/.

The 'master' is where the development work takes place and you should use this if you're interested in working with the latest cutting-edge developments. It is possible for the trunk to suffer temporary periods of instability while new features are developed. If these periods of instability are undesirable, we recommend using one of the release branches.

Chapter 2. Using Poky

This section gives an overview of the components that make up Poky followed by information about running poky builds and dealing with any problems that may arise.

2.1. Poky Overview

The BitBake task executor together with various types of configuration files form the core of Poky. This section overviews the BitBake task executor and the configuration files by describing what they are used for and they 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 a the glue to bind everything together.

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.

2.1.1. BitBake

BitBake is the tool at the heart of Poky and is responsible for parsing the metadata, generating a list of tasks from it and then executing them. To see a list of the options BitBake supports look at '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 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 glibc if they had not already been built.

2.1.2. Metadata (Recipes)

The `.bb` files are usually referred to as 'recipes'. In general, a recipe contains information about a single piece of software such as from where 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 Poky (i.e. .ipk or .deb files), this document avoids it.

2.1.3. 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.

2.1.4. Configuration

The configuration files (`.conf`) define various configuration variables that govern what Poky does. These files are split into several areas that define machine configuration options, distribution configuration options, compiler tuning options, general common configuration options and user configuration options (`local.conf`).

2.2. Running a Build

First the Poky build environment needs to be set up using the following command:

```
$ source poky-init-build-env [build_dir]
```

The build_dir is the dir containing all the build's object files. The default build dir is poky-dir/build. A different build_dir can be used for each of the targets. For example, ~/build/x86 for a qemux86 target, and ~/build/arm for a qemuarm target. Please refer to poky-init-build-env for more detailed information.

Once the Poky build environment is set up, a target can be built 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. Or, the target can be the name of a recipe for a specific piece of software such as busybox. For more details about the standard images available, 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.

2.3. Installing and Using the Result

Once an image has been built it often needs to be installed. The images/kernels built by Poky are placed in the `tmp/deploy/images` directory. Running qemux86 and qemuarm images is described in the 'Using Pre-Built Binaries and QEMU' section of the Yocto Project Quick Start. See http://www.yoctoproject.org//docs/yocto-quick-start/yocto-project-qs.html for the guide. For information about how to install these images, see the documentation for your particular board/machine.

2.4. Debugging Build Failures

The exact method for debugging Poky 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 Poky just as they are for any other system. Even though it is impossible to detail every possible potential failure, here are some general tips to aid in debugging:

2.4.1. Task Failures

The log file for shell tasks is available in `${WORKDIR}/temp/log.do_taskname.pid`. For example, the "compile" task of busybox 1.01 on the ARM spitz machine might be `tmp/work/armv5te-poky-linux-gnueabi/busybox-1.01/temp/log.do_compile.1234`. 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.

2.4.2. Running Specific Tasks

Any given package consists of a set of tasks. In most cases the series 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 - hence, the standard BitBake behaviour. 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 force option "-f". 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 WORKDIR]
$ 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. For example:

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

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

2.4.3. Dependency Graphs

Sometimes it can be hard to see why BitBake wants to build some other packages before a given package you've 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.

2.4.4. 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.

2.4.5. 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 and BitBake checks for a unique match.

2.4.6. Variables

The "-e" option dumps the resulting environment for either the configuration (no package specified) or for a specific package when specified with the "-b" option.

2.4.7. Other Tips

Tip

When adding new packages it is worth watching for undesireable 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/`.

Tip

If you want to remove the psplash boot splashscreen, add "psplash=false" to the kernel command line. Doing so prevents psplash from loading thus allowing 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. Extending Poky

This chapter provides information about how to extend the functionality already present in Poky. The chapter also documents standard tasks such as adding new software packages, extending or customizing images or porting Poky to new hardware (adding a new machine). Finally, the chapter contains advice about how to make changes to Poky to achieve the best results.

3.1. Adding a Package

To add a package into Poky 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 Recipe Variables - Required appendix.

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 Poky aims to be compatible with OpenEmbedded, most recipes should simply work in Poky.

For new packages, the simplest way to add a recipe is to base it on a similar pre-existing recipe. Following are some examples showing 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"
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 Controlling Package Content.

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 autotooled 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_2.2.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. 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"

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. Controlling Package Content

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

Following is an example that uses the "libXpm" recipe (libxpm_3.5.7.bb). By default, the "libXpm" 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"
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. 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 rootfs is made. 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 'image' directory when the rootfs is being made at build time but is unset when executed on the first boot.

3.2. Customizing Images

You can customize Poky 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-poky-x11-base package1 package2"
```

```
inherit poky-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. "glibc-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 "poky-image-sato", they can use the following recipe:

```
require poky-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

Ultimately users might want to add extra image "features" as used by Poky with the IMAGE_FEATURES variable. To create these features, the best reference is `meta/classes/poky-image.bbclass`, which shows how poky 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.

Poky 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, poky-image-sato is configured to use Dropbear. The poky-image-basic and poky-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`. 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 Poky images depend on the packages they contain. This means dependencies are automatically built when the image builds. For this reason we don't 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 poky-image-sato
```

3.3. Porting Poky to a New Machine

Adding a new machine to Poky 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 Poky already supports. Although well within the capabilities of Poky, adding a totally new architecture might require changes to `gcc/glibc` and to the site information, which is beyond the scope of this manual.

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 Poky uses to reference the new machine.

The most important variables to set in this file are TARGET_ARCH (e.g. "arm"), PREFERRED_PROVIDER_virtual/kernel (see below) and MACHINE_FEATURES (e.g. "kernel26 apm screen wifi"). You might also need other variables like SERIAL_CONSOLE (e.g. "115200 ttyS0"), KERNEL_IMAGETYPE (e.g. "zImage") and 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

Poky 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 `meta/recipes-kernel/linux` directory that can be used as references.

If you are creating a new recipe, the "normal" recipe-writing rules apply for setting up a SRC_URI. This means specifying any necessary patches and setting 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 on which Poky is running and information that Poky 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.

Reasonable defaults are used in most cases, but if customization is necessary you need to create a `machconfig` file under `meta/packages/formfactor/files/MACHINENAME/`, where `MACHINENAME` is the name for which this information applies. For information about the settings available and the defaults, see `meta/recipes-bsp/formfactor/files/config`. 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. Making and Maintaining Changes

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

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

3.4.1. BitBake Layers

Often, people want to extend Poky either by adding packages or by overriding files contained within Poky 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 Poky tree includes several additional layers such as meta-emenlow and meta-extras that demonstrate this functionality. The meta-emenlow layer is an example layer that, by default, is enabled. However, the meta-extras repository is not enabled by default. It is easy though to enable any layer. You simply add the layer's path to the BBLAYERS variable in your `bblayers.conf` file. The following example shows how to enable meta-extras in the Poky build:

```
LCONF_VERSION = "1"

BBFILES ?= ""
BBLAYERS = " \
  /path/to/poky/meta \
  /path/to/poky/meta-emenlow \
  /path/to/poky/meta-extras \
  "
```

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

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

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

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

BBFILE_COLLECTIONS += "emenlow"
BBFILE_PATTERN_emenlow := "^${LAYERDIR}/"
BBFILE_PRIORITY_emenlow = "6"
```

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 Poky tree.

Following these recommendations keeps your Poky tree and its configuration entirely inside POKYBASE.

3.4.2. Committing Changes

Modifications to Poky 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. We have found the following style works well.

Following are suggestions for committing changes to the Poky core:

- 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 for example.

- 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 policy to follow, this helps ensure the autobuilder test results are valid.

3.4.3. 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 these guidelines. First, to ensure that when a developer updates and rebuilds, they get all the changes to the repository and don't 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 Poky has upgradeable packages in all cases.

3.4.4. Using Poky in a Team Environment

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

The core component of any development effort with Poky is often an automated build testing framework and 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 people 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 poky autobuilder [http://autobuilder.pokylinux.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. They 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 Poky (i.e. git or 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. Poky easily supports testing configurations that use both a stable known good revision and a floating revision. Poky 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 Poky metadata policies that surround the different source control systems. Of course circumstances will be different in each case. However, this situation reveals one of Poky'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.

3.4.5. 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
```

3.5. Modifying Package Source Code

Although Poky 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.

3.5.1. 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 Poky 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. Track License Change

The license of an upstream project might change in the future. Poky uses the LIC_FILES_CHKSUM variable to track license changes.

3.6.1. Specifying the LIC_FILES_CHKSUM Variable

The LIC_FILES_CHKSUM variable contains checksums of the license text in the recipe source code. Poky uses this to track changes in the license text of the source code files. Following is an example of 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 \
                    ..."
```

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

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

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

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

3.6.2. 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. Using this variable you can specify the line on which the license text starts and ends by supplying "beginline" and "endline" parameters. 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 as 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 mis-match 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 mis-match occurs during the build, the correct md5 checksum is placed in the build log and can be easily copied to a .bb file.

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 allow 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.

3.7. Handling 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. Poky 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.

3.7.1. Specifying 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 Poky 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 Poky recipes.

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

The results are stored in the `build/tmp/log/distro_check-${DATETIME}.results` file.

Chapter 4. Board Support Packages (BSP) - Developers 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.

The proposed format does have elements that are specific to the Poky and OpenEmbedded build systems. It is intended that this information can be used by other systems besides Poky and OpenEmbedded and that it will be simple to extract information and convert it to other formats if required. Poky, 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.

4.1. 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-intel_n450
meta-beagleboard
```

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_git.bbappend
```

Below is an example of the crownbay BSP:

```
meta-crownbay/COPYING.MIT
meta-crownbay/README
meta-crownbay/binary/.gitignore
meta-crownbay/conf/layer.conf
meta-crownbay/conf/machine/crownbay.conf
meta-crownbay/recipes-bsp/formfactor/formfactor/crownbay/machconfig
meta-crownbay/recipes-bsp/formfactor/formfactor_0.0.bbappend
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config/crownbay/xcorg.conf
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config_0.1.bbappend
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-emgd-bin/.gitignore
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-emgd-bin_1.7.99.2.bb
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-emgd/crosscompile.patch
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-emgd/fix_open_max_preprocessor_error.pat
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-emgd/macro_tweak.patch
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-emgd/nodolt.patch
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-emgd_1.7.99.2.bb
meta-crownbay/recipes-kernel/linux/linux-yocto_git.bbappend
```

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

4.1.1. License Files

```
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 crownbay 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.

4.1.2. README File

```
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.

4.1.3. Pre-built User Binaries

```
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 contains the Application Development Toolkit (ADT) and minimal live images when the BSP is has been "tar-balled" and placed on 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.

4.1.4. Layer Configuration File

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

This file identifies the structure as a Poky layer, identifies the contents of the layer, and contains information about how Poky 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 Poky can recognize the BSP.

4.1.5. Hardware Configuration Options

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

The machine files bind together all the information contained elsewhere in the BSP into a format that Poky 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.

This directory could also contain shared hardware "tuning" definitions that are commonly used to pass specific optimization flags to the compiler. An example is `tune-atom.inc`:

```
BASE_PACKAGE_ARCH = "core2"
TARGET_CC_ARCH = "-m32 -march=core2 -msse3 -mtune=generic -mfpmath=sse"
```

This example defines a new package architecture called "core2" and uses the specified optimization flags, which are carefully chosen to give best performance on atom processors.

The tune file would be included by the machine definition and can be contained in the BSP or referenced from one of the standard core set of files included with Poky itself.

Both the base package architecture file and the tune file are optional for a Poky BSP layer.

4.1.6. Miscellaneous Recipe Files

`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 crownbay BSP there is a `machconfig` file and a `formfactor_0.0.bbappend` file:

```
meta-crownbay/recipes-bsp/formfactor/formfactor/crownbay/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.

4.1.7. Display Support Files

```
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, in the crownbay BSP several display support files exist:

```
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config/crownbay/xcorg.conf
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-config_0.1.bbappend
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-emgd-bin/.gitignore
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-emgd-bin_1.7.99.2.bb
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-emgd/crosscompile.patch
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-emgd/fix_open_max_preprocessor_error.pat
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-emgd/macro_tweak.patch
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-emgd/nodolt.patch
meta-crownbay/recipes-graphics/xorg-xserver/xserver-xf86-emgd_1.7.99.2.bb
```

4.1.8. Linux Kernel Configuration

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

This file appends your specific changes to the kernel you are using.

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

Suppose you use a BSP that uses the `linux-yocto_git.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 statement:

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
```

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

Now take a look at the existing "crownbay" BSP. The append file used is:

```
meta-crownbay/recipes-kernel/linux/linux-yocto_git.bbappend
```

The file contains the following:

```
FILESEXTRAPATHS := "${THISDIR}/${PN}"
COMPATIBLE_MACHINE_crownbay = "crownbay"
```

```
KMACHINE_crownbay = "yocto/standard/crownbay"
```

This append file adds "crownbay" as a compatible machine, and additionally sets a Yocto Kernel-specific variable that identifies the name of the BSP branch to use in the GIT repository to find configuration information.

One thing missing in this particular BSP, which you will typically need when developing a BSP, is the kernel configuration (.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" on the Yocto Project website Documentation Page [http://yoctoproject.org/community/documentation]

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.

4.2. 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 Poky 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 at http://yoctoproject.org/download/board-support-package-bsp-downloads will make available a 'de-featured' BSP completely free of the encumbered IP. In that case you can use the substitution directly and without any further licensing requirements. If present, this fully 'de-featured' BSP will be named appropriately different than the normal encumbered BSP. If available, this substitution is the simplest and most preferred option. 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 Poky 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 poky-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.

   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 `poky/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 5. Platform Development with Poky

5.1. Software development

Poky supports several methods of software development. You can use the method that is best for you. This chapter describes each development method.

5.1.1. External Development Using the Poky SDK

The meta-toolchain and meta-toolchain-sdk targets build tarballs that contain toolchains and libraries suitable for application development outside of Poky. For information on these targets see the Reference: Images appendix.

These tarballs unpack into the `/opt/poky` directory and contain a setup script (e.g. `/opt/poky/environment-setup-i586-poky-linux`), from which you can source to initialize a suitable environment. Sourcing these files adds the compiler, QEMU scripts, QEMU binary, a special version of pkgconfig and other useful utilities to the PATH variable. 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 other projects it is usually a case of ensuring the cross tools are used:

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

5.1.2. Using the Eclipse and Anjuta Plug-ins

Yocto Project supports both Anjuta and Eclipse IDE plug-ins to make developing software easier for the application developer. The plug-ins provide capability extensions to the graphical IDE allowing for cross compilation, deployment and execution of the output in a QEMU emulation session. Support of these plug-ins 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 data, collection of latency data and collection of performance data.

5.1.2.1. The Eclipse Plug-in

To use the Eclipse plug-in, a toolchain and SDK built by Poky is required along with the Eclipse Framework (Helios 3.6.1). To install the plug-in you need to be in the Eclipse IDE and select the following menu:

```
Help -> Install New Software
```

Specify the target URL as http://www.yoctoproject.org/downloads/eclipse-plugin/.

If you want to download the source code for the plug-in you can find it in the Poky git repository, which has a web interface, and is located at http://git.yoctoproject.org under IDE Plugins.

5.1.2.1.1. Installing and Setting up the Eclipse IDE

If you don't have the Eclipse IDE (Helios 3.6.1) on your system you need to download and install it from http://www.eclipse.org/downloads. Choose the Eclipse Classic, which contains the Eclipse Platform, Java Development Tools (JDT), and the Plug-in Development Environment.

Note

Due to the Java Virtual Machine's garbage collection (GC) process the permanent generation space (PermGen) is not cleaned up. This space stores meta-data descriptions of classes. The default value is set too small and it could trigger an out-of-memory error like the following:

```
Java.lang.OutOfMemoryError: PermGen space
```

This error causes the applications to hang.

To fix this issue you can use the `-vmargs` option when you start Eclipse to increase the size of the permanent generation space:

```
Eclipse -vmargs -XX:PermSize=256M
```

5.1.2.1.2. Installing the Yocto Plug-in

Once you have the Eclipse IDE installed and configured you need to install the Yocto plug-in. You do this similar to installing the Eclipse plug-ins in the previous section.

Do the following to install the Yocto plug-in into the Eclipse IDE:

1. Select the "Help -> Install New Software" item.
2. In the "Work with:" area click "Add..." and enter the URL for the Yocto plug-in, which is http://www.yoctoproject.org/downloads/eclipse-plugin/
3. Finish out the installation of the update similar to any other Eclipse plug-in.

5.1.2.1.3. Configuring Yocto Eclipse plug-in

To configure the Yocto Eclipse plug-in you need to select the mode and the architecture with which you will be working. Start by selecting "Preferences" from the "Window" menu and then select "Yocto SDK".

If you normally will use an installed Yocto SDK (under `/opt/poky`) select “SDK Root Mode”. Otherwise, if your crosstool chain and sysroot are within your poky tree, select “Poky Tree Mode”. If you are in SDK Root Mode you need to provide your poky tree path, for example, `$<Poky_tree>/build/`.

Next, you need to select the architecture. Use the drop down list and select the architecture that you’ll be primarily working against. For target option, select your typical target QEMU vs External hardware. If you choose QEMU, you’ll need to specify your QEMU kernel file with full path and the rootfs mount point. Yocto QEMU boots off user mode NFS. See the Developing Externally in QEMU section for how to set it up.

To make your settings the defaults for every new Yocto project created using the Eclipse IDE, simply save the settings.

5.1.2.1.4. Using the Yocto Eclipse Plug-in

As an example, this section shows you how to cross-compile a Yocto C project that is autotools-based, deploy the project into QEMU, and then run the debugger against it. You need to configure the project, trigger the `autogen.sh`, build the image, start QEMU, and then debug.

The following steps show how to create a Yocto autotools-based project using a given template:

1. Select "File -> New -> Project" to start the wizard.
2. Expand "C/C++" and select "C Project".
3. Click "Next" and select a template (e.g. "Hello World ANSI C Project").
4. Complete the steps to create the new Yocto autotools-based project using your chosen template.

By default, the project uses the Yocto preferences settings as defined using the procedure in the previous section. If there are any specific setup requirements for the newly created project you need to reconfigure the Yocto plug-in through the menu selection by doing the following:

1. Select the "Project -> Invoke Yocto Tools -> Reconfigure Yocto" menu item.

2. Complete the dialogue to specify the specific toolchain and QEMU setup information.

To build the project follow these steps:

1. Select "Project -> Reconfigure Project" to trigger the `autogen.sh` command.

2. Select "Project -> Build" to build the project.

To start QEMU follow these steps:

1. Select "Run -> External Tools" and see if there is a QEMU instance for the desired target. If one exists, click on the instance to start QEMU. If your target does not exist, click "External Tools Configuration" and you should find an instance of QEMU for your architecture under the entry under "Program".

2. Wait for the boot to complete.

To deploy your project and start debugging follow these steps:

1. Highlight your project in the project explorer.

2. Select "Run -> Debug Configurations" to bring up your remote debugging configuration in the right-hand window.

3. Expand “C/C++ Remote Application”.

4. Select "projectname_ gdb_target-poky-linux". You need to be sure there is an entry for the remote target. If no entry exists, click "New..." to bring up the wizard. Use the wizard to select TCF and enter the IP address of you remote target in the “Host name:” field. Back in the Remote Debug Configure window, specify in the “Remote Absolute File Path for C/C++ Application” field the absolute path for the program on the remote target. By default, the program deploys into the remote target. If you don't want this behavior then check “Skip download to target path”.

5. Click "Debug” to start the remote debugging session.

5.1.2.1.5. Using Yocto Eclipse plug-in Remote Tools Suite

Remote tools allow you to perform system profiling, kernel tracing, examine power consumption, and so forth. To see and access the remote tools use the "Window -> YoctoTools" menu.

Once you pick a tool you need to configure it for the remote target. Every tool needs to have the connection configured. You must select an existing TCF-based RSE connection to the remote target. If one does not exist, click "New" to create one.

Here are some specifics about the remote tools:

- OProfile: Selecting this tool causes the oprofile-server on the remote target to launch on the local host machine. The oprofile-viewer must be installed on the local host machine and the oprofile-server must be installed on the remote target, respectively, in order to use .

- lttng: Selecting this tool runs "usttrace" on the remote target, transfers the output data back to the local host machine and uses "lttv-gui" to graphically display the output. The "lttv-gui" must be installed on the local host machine to use this tool. For information on how to use "lttng" to trace an application, see http://lttng.org/files/ust/manual/ust.html.

  For "Application" you must supply the absolute path name of the application to be traced by user mode lttng. For example, typing `/path/to/foo"` triggers "usttrace /path/to/foo" on the remote target to trace the program `/path/to/foo`.

  "Argument" is passed to "usttrace" running on the remote target.

- powertop: Selecting this tool runs "powertop" on the remote target machine and displays the results in a new view called "powertop".

"Time to gather data(sec):" is the time passed in seconds before data is gathered from the remote target for analysis.

"show pids in wakeups list:" corresponds to the `-p` argument passed to "powertop".

- latencytop and perf: "latencytop" identifies system latency, while "perf" monitors the system's performance counter registers. Selecting either of these tools causes an RSE terminal view to appear from which you can run the tools. Both tools refresh the entire screen to display results while they run.

5.1.2.2. The Anjuta Plug-in

Note

Support for the Anjuta plug-in ends after Yocto project 0.9 Release. However, the source code can be downloaded from the git repository listed later in this section. The community is free to continue supporting it post 0.9 Release.

An Anjuta IDE plug-in exists to make developing software within the Poky framework easier for the application developer familiar with that environment. The plug-in presents a graphical IDE that allows you to cross-compile, cross-debug, profile, deploy, and execute an application.

To use the plug-in, a toolchain and SDK built by Poky, Anjuta, its development headers and the Anjuta Plug-in are all required. The Poky Anjuta Plug-in is available to download as a tarball at the OpenedHand labs http://labs.o-hand.com/anjuta-poky-sdk-plugin/ page or directly from the Poky Git repository located at git://git.yoctoproject.org/anjuta-poky. You can access the source code from a web interface to the repository at http://git.yoctoproject.org/ under IDE Plugins.

See the README file contained in the project for more information on Anjuta dependencies and building the plug-in. If you want to disable remote gdb debugging, pass the "--disable-gdb-integration" switch when you configure the plug-in.

5.1.2.2.1. Setting Up the Anjuta Plug-in

Follow these steps to set up the plug-in:

1. Extract the tarball for the toolchain into / as root. The toolchain will be installed into `/opt/poky`.

2. To use the plug-in, first open or create an existing project. If you are creating a new project, the "C GTK+" project type will allow itself to be cross-compiled. However, you should be aware that this type uses "glade" for the UI.

3. To activate the plug-in, select "Edit -> Preferences" and then choose "General" from the left hand side. Choose the "Installed plug-ins" tab, scroll down to "Poky SDK" and check the box.

The plug-in is now activated but not configured.

5.1.2.2.2. Configuring the Anjuta Plug-in

You can find the configuration options for the SDK by choosing the Poky SDK icon from the left hand side. You need to define the following options:

- SDK root: If you use an external toolchain you need to set SDK root, which is the root directory of the SDK's sysroot. For an i586 SDK directory is `/opt/poky/`. This directory will contain "bin", "include", "var" and so forth under your selected target architecture subdirectory `/opt/poky/sysroot/i586-poky-linux/`. The cross-compile tools you need are in `/opt/poky/sysroot/i586-pokysdk-linux/`.

- Poky root: If you have a local Poky build tree, you need to set the Poky root, which is the root directory of the poky build tree. If you build your i586 target architecture under the subdirectory of `build_x86` within your Poky tree, the Poky root directory should be `$<poky_tree>/build_x86/`.

- Target Architecture: This is the cross compile triplet, for example, "i586-poky-linux". This target triplet is the prefix extracted from the set up script file's name. For example, if the script file name is

`/opt/poky/environment-setup-i586-poky-linux` then the extracted target triplet is "i586-poky-linux".

- Kernel: Use the file chooser to select the kernel used with QEMU.
- Root filesystem: Use the file chooser to select the root filesystem directory. This directory is where you use "poky-extract-sdk" to extract the poky-image-sdk tarball.

5.1.2.2.3. Using the Anjuta Plug-in

The steps in this section show how to cross-compile a project, deploy it into QEMU, run a debugger against it and then perform a system-wide profile.

1. Choose "Build -> Run Configure" or "Build -> Run Autogenerate" to run "configure" or "autogen", respectively for the project. Either command passes command-line arguments to instruct the cross-compile.
2. Choose "Build -> Build Project" to build and compile the project. If you have previously built the project in the same tree without using the cross-compiler you might find that your project fails to link. If this is the case, simply select "Build -> Clean Project" to remove the old binaries. After you clean the project you can then try building it again.
3. Choose "Tools -> Start QEMU" to start QEMU. After QEMU starts any error messages will appear in the message view. Once Poky has fully booted within QEMU you can deploy the project into it.
4. Once the project is built and you have QEMU running choose "Tools -> Deploy" to install the package into a temporary directory and then copy it using "rsync" over SSH into the target. A progress bar and appropriate messages appear in the message view.
5. To debug a program installed onto the target choose "Tools -> Debug remote". Choosing this menu item causes prompts to appear to define the local binary for debugging and also for the command line used to run on the target. When you provide the command line be sure to include the full path to the to binary installed in the target. When the command line runs a "gdbserver" over SSH is started on the target and an instance of "cross-gdb" starts in a local terminal. The instance of "cross-gdb" will be preloaded to connect to the server and use the SDK root to find symbols. It also connects to the target and loads in various libraries as well as the target program. You should define any breakpoints or watchpoints at this point in the process since you might not be able to interrupt the execution later. To stop the debugger on the target choose "Tools -> Stop debugger".
6. It is also possible to execute a command in the target over SSH. Doing so causes the appropriate environment to be established for execution. To execute a command choose "Choose Tools -> Run remote". This selection opens a terminal with the SSH command inside.
7. To perform a system-wide profile against the system running in QEMU choose "Tools -> Profile remote". This choice starts up "OProfileUI" with the appropriate parameters to connect to the server running inside QEMU and also supplies the path for debug information necessary to get a useful profile.

5.1.3. Developing Externally in QEMU

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

Poky's QEMU images contain a complete native toolchain. This means you can develop applications within QEMU similar to the way you would in a normal system. Using qemux86 on an x86 machine is fast since the guest and host architectures match. On the other hand, using qemuarm can be slower but gives faithful emulation of ARM-specific issues. To speed things up, these images support using "distcc" to call a cross-compiler outside the emulated system. If "runqemu" was used 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` it is also automatically be used.

There are several options for connecting into the emulated system. QEMU provides a framebuffer interface that has standard consoles available. There is also a serial connection available that has a console to the system running on it and uses standard IP networking. The images have a dropbear

ssh server running with the root password disabled to allow standard ssh and scp commands to work. The images also contain an NFS server that exports the guest's root filesystem, which allows it to be made available to the host.

5.1.4. Developing in Poky Directly

Working directly in Poky is a fast and effective development technique. The idea is that you can directly edit files in 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
$ exit
$ bitbake matchbox-desktop -c compile -f
$ bitbake matchbox-desktop
```

This example builds the package, changes into the work directory for the package, changes a file, then recompiles the package. Instead of using "sh" as it is in the example, you can also use two different terminals. However, the risk of using two terminals is that a command like "unpack" could destroy the changes you've made to 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 "quilt" as detailed in the modifying packages with quilt section. You can copy the resulting patches into the recipe directory and use them directly in the SRC_URI.

For a review of the skills used in this section see the Bitbake and Running Specific Tasks Sections.

5.1.5. Developing with 'devshell'

When debugging certain commands or even when just editing packages, the 'devshell' can be a useful tool. Use a command like the following to start this tool.

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

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

- The PATH variable includes the cross toolchain.
- The pkgconfig variables find the correct `.pc` files.
- "configure" finds the Poky 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 Poky itself. The working directory also automatically changes to the (S) directory. When you are finished, you just exit the shell or close the terminal window.

The default shell used by "devshell" is the gnome-terminal. You can use other forms of terminal can by setting the TERMCMD and TERMCMDRUN variables in `local.conf`. For examples of the other options available, see `meta/conf/bitbake.conf`.

An external shell is launched rather than opening directly into the original terminal window. This allows easier interaction with Bitbake's multiple threads as well as for a future client/server split. Note that "devshell" will still work over X11 forwarding or similar situations.

It is worth remembering that inside "devshell" you need to use the full compiler name such as `arm-poky-linux-gnueabi-gcc` instead of just `gcc`. The same applies to other applications such as gcc, bintuils, libtool and so forth. Poky will have setup environmental variables such as CC to assist applications, such as make, find the correct tools.

5.1.6. Developing within Poky with an External SCM-based Package

If you're working on a recipe that pulls from an external SCM it is possible to have Poky notice new changes added to the SCM and then build the latest version using them. This only works for SCMs from which it is possible to get a sensible revision number for changes. Currently it works for svn, git and bzr repositories.

To enable this behavior simply add SRCREV_pn- PN = "${AUTOREV}" to `local.conf`, where PN is the name of the package for which you want to enable automatic source revision updating.

5.2. Debugging with GDB Remotely

GNU Project Debugger (GDB) allows you to examine running programs to understand and fix problems and also to perform post-mortem style analysis of program crashes. GDB is available as a package within Poky and by default is installed in sdk images. See http://sourceware.org/gdb/ for the GDB source.

Tip

For best results install `-dbg` packages for the applications you are going to debug. Doing so makes available extra debug symbols that will 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 these constraints you can use GDBSERVER. It 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 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 at on their site [http://sourceware.org/gdb/documentation/].

5.2.1. Launching GDBSERVER on the Target

First, make sure GDBSERVER is installed on the target. If 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.

5.2.2. Launching GDB on the Host Computer

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

5.2.2.1. 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 Poky toolchain - for 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, Poky can build the `gdb-cross` binary. For example, the following command builds it:

```
$ bitbake gdb-cross
```

Once the binary is built you can find it here:

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

5.2.2.2. 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 is to have an 'sdk' image that corresponds to the plain image installed on the device. In the case of 'poky-image-sato', 'poky-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, Poky 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 Poky.
- 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.

These 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>/poky-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>/poky-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>/poky-image-sato-1.0-r0/temp/opkg.conf \
-o tmp/rootfs install foo-dbg
```

5.2.2.3. 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.

5.2.2.4. 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/.

5.3. 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 Poky 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 IMAGE_FEATURES. 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 IMAGE_FEATURES 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 Poky you can achieve this by setting SELECTED_OPTIMIZATION to "-fexpensive-optimizations -fno-omit-framepointer -frename-registers

-O2". You can also achieve it by setting DEBUG_BUILD to "1" in `local.conf`. If you use the DEBUG_BUILD variable you will also add extra debug information that can make the debug packages large.

5.3.1. 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 `/boot/` in Poky 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/.

5.3.2. Using OProfileUI

A graphical user interface for OProfile is also available. You can download and build it from svn at http://svn.o-hand.com/repos/oprofileui/trunk/. If the "tools-profile" image feature is selected, all necessary binaries are installed onto the target device for OProfileUI interaction.

In order to convert the data in the sample format from the target to the host you need the `opimport` program. This program is not included in standard Debian OProfile packages. However, an OProfile package with this addition is available from the OpenedHand repository [http://debian.o-hand.com/]. We recommend using OProfile 0.9.3 or greater.

Even though Poky 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://svn.o-hand.com/repos/oprofileui/trunk/README] for the most recent information. You can also see OProfileUI website [http://labs.o-hand.com/oprofileui] for general information on the OProfileUI project.

5.3.2.1. 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. The buttons are:

- 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 `/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 Poky, that file is usually located in `/boot/vmlinux-KERNELVERSION`, where KERNEL-version is the version of the kernel (e.g. 2.6.23). Poky generates separate vmlinux packages for each kernel it builds so it should be a question of just making sure a matching package is installed - for example: `opkg install kernel-vmlinux`. The files are automatically installed into development and profiling images alongside OProfile. There is a configuration option within the OProfileUI settings page where you can 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 Poky 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.

5.3.2.2. 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

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

A.1. Top level core components

A.1.1. **bitbake/**

Poky 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 Poky 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 poky-init-build-env script.

For more information on BitBake, see the BitBake project site at http://bitbake.berlios.de/ and 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 Poky in its standard configuration where the source tree is combined with the output. It is also possible to place output and configuration files in a directory separate from the Poky source. For information on separating output from the Poky source, see poky-init-build-env.

A.1.3. **meta/**

This directory contains the core metadata, which is a key part of Poky. This directory contains the machine definitions, the Poky distribution, and the packages that make up a given system.

A.1.4. **meta-extras/**

This directory is similar to `meta/`. The directory contains extra metadata not included in standard Poky. This metadata is disabled by default and is not supported as part of Poky.

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

These directories are optional layers that are added to core metadata. The layers are enabled by adding them to the `conf/bblayers.conf` file.

A.1.6. **scripts/**

This directory contains various integration scripts that implement extra functionality in the Poky environment (e.g. QEMU scripts). The poky-init-build-env script appends this directory to the PATH environment variable.

A.1.7. **sources/**

This directory receives downloads as specified by the DL_DIR variable. Even though the directory is not part of a checkout, Poky creates it during a build. You can use this directory to share downloading files between Poky builds. This practice can save you from downloading files multiple times.

> **Note**
>
> You can override the location for this directory by setting the DL_DIR variable in `local.conf`.

This directory also contains SCM checkouts (e.g. `sources/svn/` , `sources/cvs/` or `sources/git/`). The `sources` directory can contain archives of checkouts for various revisions or dates.

It's worth noting that BitBake creates `.md5` stamp files for downloads. BitBake uses these files to mark downloads as complete as well as for checksum and access accounting purposes. If you manually add a file to the directory, you need to touch the corresponding `.md5` file as well.

A.1.8. **documentation**

This directory holds the source for the documentation. Each manual is contained in a sub-folder. For example, the files for this manual reside in `poky-ref-manual`.

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

This script sets up the Poky build environment. Sourcing this file 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 Poky commands. The script uses other scripts within the `scripts/` directory to do the bulk of the work. You can use this script to specify any directory for the build's output by doing the following:

```
$ source POKY_SRC/poky-init-build-env [BUILDDIR]
```

You can enter the above command from any directory, as long as POKY_SRC points to the desired Poky source tree. The optional BUILDDIR can be any directory into which you would like Poky to generate the build output.

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

A.2.1. **build/conf/local.conf**

This file contains all the local user configuration of Poky. 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 Poky unless that variable is hard-coded within Poky (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.2.2. **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 the environment setup script is sourced.

A.2.3. **build/tmp/**

This directory receives all the Poky output. BitBake creates this directory if it does not exist. To clean Poky and start a build from scratch (other than downloads), you can remove everything in this directory or get rid of the directory completely. The `tmp/` directory has some important sub-components detailed below.

A.2.4. **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.2.5. **build/tmp/deploy/**

This directory contains any 'end result' output from Poky.

A.2.6. **build/tmp/deploy/deb/**

This directory receives any .deb packages produced by Poky. The packages are sorted into feeds for different architecture types.

A.2.7. **build/tmp/deploy/rpm/**

This directory receives any .rpm packages produced by Poky. The packages re sorted into feeds for different architecture types.

A.2.8. **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.

A.2.9. **build/tmp/deploy/ipk/**

This directory receives .ipk packages produced by Poky.

A.2.10. **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.2.11. **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.2.12. **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.2.13. **build/tmp/pkgdata/**

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

A.2.14. **build/tmp/pstagelogs/**

This directory contains manifest for task-based pre-built. Each manifest is basically a file list for installed files from a given task. Manifests are useful for later packaging or cleanup processes.

A.2.15. **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.

It is worth considering the structure of a typical work directory. As an example consider the linux-rp kernel, version 2.6.20 r7 on the machine spitz built within Poky. For this package a work directory of `tmp/work/spitz-poky-linux-gnueabi/linux-rp-2.6.20-r7/`, referred to as WORKDIR, is created. Within this directory, the source is unpacked to linux-2.6.20 and then patched by quilt (see Section 3.5.1). Within the `linux-2.6.20` directory, standard quilt directories `linux-2.6.20/patches` and `linux-2.6.20/.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 Poky. 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 Poky 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 includes 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="spitz", Poky looks for a `spitz.conf` file in this directory. The includes directory contains various data common to multiple machines. If you want to add support for a new machine to Poky, look in this directory.

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

Any distribution-specific configuration is controlled from this directory. Poky only contains the Poky distribution so `poky.conf` is the main file here. This directory includes the versions and SRCDATES for applications that are configured here. An example of an alternative configuration is `poky-bleeding.conf` although this file mainly inherits its configuration from Poky itself.

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

This directory contains anything linking to specific hardware or hardware configuration information such as "uboot" 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/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.

Appendix B. Reference: BitBake

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

```
$ bitbake poky-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. Poky keeps this file in `meta/conf/`. BitBake finds it by examining the BBPATH environment variable and looking for the `meta/conf/` directory.

In Poky, `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 Poky. 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 inculded. 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/packages/` directory within Poky, but other directories such as `meta-extras/` can be included too. Adding extra content to BBFILES is best achieved through the use of BitBake "layers".

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 (poky-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 "poky-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 glibc and the toolchain.

Sometimes a target might have multiple providers. An common example is "virtual/kernel", which is provided by each kernel package. Each machine often elects the best kernel provider by using a line similar to the following in the machine configuration file:

```
PREFERRED_PROVIDER_virtual/kernel = "linux-rp"
```

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 taksks have all their dependencies met, and the thread threshold has not been exceeded.

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.

B.6. BitBake Command Line

Following is the bitbake manpage:

```
$ 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.
  -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
  -i, --interactive     drop into the interactive mode also called the BitBake
                        shell.
  -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 FILE, --read=FILE  read the specified file before 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
  -p, --parse-only      quit after parsing the BB files (developers only)
  -d, --disable-psyco   disable using the psyco just-in-time compiler (not
                        recommended)
  -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 IGNORED_DOT_DEPS, --ignore-deps=IGNORED_DOT_DEPS
                        Stop processing at the given list of dependencies when
                        generating dependency graphs. This can help to make
                        the graph 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
```

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 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 Developing within Poky with an External SCM-based Package 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 or the directory pointed by BUILDDIR (e.g. `build/`)in the same way as `.conf` files in the `conf` directory. Class files are searched for in BBPATH in the same was as `.conf` files too.

In most cases inheriting the class is enough to enable its features, although for some classes you may need to set variables and/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. It contains definitions of 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 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, libtool) brings standardisation and this class aims to define a set of tasks (configure, compile etc.) that will work for all autotooled packages. It should usually be enough to define a few standard variables as documented in the simple autotools example 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. Additional parameters can be passed to configure through the EXTRA_OECONF variable.
- 'do_compile' runs make with arguments specifying the compiler and linker. Additional arguments can be passed through the EXTRA_OEMAKE variable.
- 'do_install' runs make install passing a DESTDIR option taking its value from the standard DESTDIR variable.

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

Several programs can fulfill the same or similar function and they can be installed with the same name. For example the ar command is available from the "busybox", "binutils" and "elfutils" packages. This class handles the renaming of the binaries so multiple packages can be installed which would otherwise conflict and yet the ar command still works regardless of which are installed or subsequently removed. It 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	Name of binary which will be replaced (ar in this example)
ALTERNATIVE_LINK	Path to resulting binary ("/bin/ar" in this example)
ALTERNATIVE_PATH	Path to real binary ("/usr/bin/ar.binutils" in this example)
ALTERNATIVE_PRIORITY	Priority of binary, the version with the most features should have the highest priority

Currently, only one binary per package is supported.

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

This class uses update-rc.d to safely install an initscript on behalf of the package. Details such as making sure the initscript is stopped before a package is removed and started when the package is installed are taken care of. Three variables control this class, INITSCRIPT_PACKAGES, INITSCRIPT_NAME and INITSCRIPT_PARAMS. See the 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. It also changes all paths to point into the `sysroots/` directory so all builds which 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. 'glibc' becomes 'libc6' and 'glibc-devel' becomes 'libc6-dev'.

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

Pkg-config brought standardisation and this class aims to make its integration smooth for all libraries which make use of it.

During staging Bitbake installs pkg-config data into the `sysroots/` directory. By making use of sysroot functionality within pkgconfig this class no longer has to manipulate the files.

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

Many software licenses require providing the sources for compiled binaries. To simplify this process two classes were created: `src_distribute.bbclass` and `src_distribute_local.bbclass`.

Result of their work are `tmp/deploy/source/` subdirs with sources sorted by LICENSE field. If recipe lists few licenses (or has entries like "Bitstream Vera") source archive is put in each license dir.

Src_distribute_local class has three modes of operating:

- copy - copies the files to the distribute dir
- symlink - symlinks the files to the distribute dir
- move+symlink - moves the files into distribute dir, and symlinks them back

C.9. Perl modules -**cpan.bbclass**

Recipes for Perl modules are simple - usually needs only pointing to source archive and inheriting of proper bbclass. Building is split into two methods dependly on method used by module authors.

Modules which use old Makefile.PL based build system require using of `cpan.bbclass` in their recipes.

Modules which use Build.PL based build system require using of `cpan_build.bbclass` in their recipes.

C.10. Python extensions -**distutils.bbclass**

Recipes for Python extensions are simple - they usually only require pointing to the source archive and inheriting the proper bbclasses. Building is split into two methods depending on the build method used by the module authors.

Extensions which use autotools based build system require use of autotools and distutils-base bbclasses in their recipes.

Extensions which use distutils build system require use of `distutils.bbclass` in their recipes.

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

This class adds the devshell task. Its usually up to distribution policy to include this class (Poky does). See the developing with 'devshell' section for more information about using devshell.

C.12. Packaging -**package*.bbclass**

The packaging classes add support for generating packages from a builds output. The core generic functionality is in `package.bbclass`, 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 and this is controlled by the PACKAGE_CLASSES variable. The first class listed in this variable will be used for image generation. Since images are generated from packages a packaging class is needed to enable image generation.

C.13. Building kernels -**kernel.bbclass**

This class handles building of Linux kernels and the class contains code to know how to build both 2.4 and 2.6 kernel trees. All needed headers are staged into STAGING_KERNEL_DIR directory to allow building of out-of-tree modules using `module.bbclass`.

This means that each kernel module built 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 will install 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**

Those classes add support for creating images in many formats. First the rootfs is created from packages by one of the `rootfs_*.bbclass` files (depending on package format used) and then image is created. The IMAGE_FSTYPES variable controls which types of image to generate. The list of packages to install into the image is controlled by the IMAGE_INSTALL variable.

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

This class checks prerequisite software is present to notify the users of potential problems that will affect their build. It also performs basic checks of the user configuration from local.conf to prevent common mistakes resulting in build failures. It's usually up to distribution policy whether to include this class (Poky does).

C.16. Generated output quality assurance checks - **insane.bbclass**

This class adds a step to package generation which sanity checks the packages generated by Poky. There are an ever increasing range of checks it performs, checking for common problems which break builds/packages/images, see the bbclass file for more information. It's usually up to distribution policy whether to include this class (Poky does).

C.17. Autotools configuration data cache - **siteinfo.bbclass**

Autotools can require tests which have to execute on the target hardware. Since this isn't possible in general when cross compiling, siteinfo is used to provide cached test results so these tests can be skipped over but the correct values used. The meta/site directory contains test results sorted into different categories like architecture, endianess and the libc used. Siteinfo provides a list of files containing data relevant to the current build in the CONFIG_SITE variable which autotools will automatically pick up.

The class also provides variables like SITEINFO_ENDIANESS and SITEINFO_BITS which can be used elsewhere in the metadata.

This class is included from `base.bbclass` and is hence always active.

C.18. Other Classes

Only the most useful/important classes are covered here but there are others, see the `meta/classes` directory for the rest.

Appendix D. Reference: Images

Poky has several standard images covering most people's standard needs. Use the following command to list the supported images:

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

Images are listed below along with details of what they contain:

> ## 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 IMAGE_EXTRA_FEATURES line
> 2. Set INCOMPATIBLE_LICENSE = "GPLv3"

- poky-image-minimal - A small image just capable of allowing a device to boot.
- poky-image-base - A console-only image that fully supports the target device hardware.
- poky-image-core - An X11 image with simple applications such as terminal, editor, and file manager.
- poky-image-sato - An X11 image with Sato theme and Pimlico applications. The image also contains terminal, editor, and file manager.
- poky-image-sato-dev - An X11 image similar to poky-image-sato but 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 poky-image-sdk.
- poky-image-lsb - An image suitable for implementations that conform to Linux Standard Base (LSB).
- meta-toolchain - This image generates a tarball that contains a stand-alone toolchain that can be used externally to Poky. The tarball is self-contained and unpacks to the `/opt/poky` directory. The tarball also contains a copy of QEMU and the scripts necessary to run poky QEMU images.
- meta-toolchain-sdk - This image 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.

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 distribution configuration file (poky.conf for Poky). Machine features are set in the MACHINE_FEATURES variable which is set in the machine configuration file and specifies which hardware features a given machine has.

These two variables are combined 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 doesn't 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 Poky can be controlled by the IMAGE_FEATURES variable in local.conf. Through this 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:

- 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, file editor
- apps-x11-games - A set of X11 games
- apps-x11-pimlico - OpenedHand Pimlico application suite
- tools-sdk - A full SDK which runs on 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 Poky 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	E-mail address to contact original author(s) - to send patches, forward bugs...
AUTOREV	Use current (newest) source revision - used with SRCREV variable.

B

BB_NUMBER_THREADS	The maximum number of tasks BitBake should run in parallel at any one time
BBFILE_COLLECTIONS	Identifies layer-specific bbfiles, which contain recipes used by BitBake to build software. The Variable is appended with a layer name.
BBFILE_PATTERN	Variable that expands to match files from BBFILES in a particular layer. BBFILE_PATTERN is used in the `conf/layer.conf` file and must contain the name of the specific layer (e.g. BBFILE_PATTERN_emenlow).
BBFILE_PRIORITY	Assigns different priorities to recipe files in different layers. This variable is useful in situations where the same package might appear in multiple layers. It allows you to choose what takes precedence.
BBFILES	List of recipes 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 which controls how BitBake displays logs on build failure.
BBLAYERS	Lists in the `bblayers.conf` file layers to enable in the Poky build.
BPN	Bare name of package with any suffixes like -cross -native removed.

C

CFLAGS	Flags passed to C compiler for the target system. 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 they're incompatible with, which is particularly useful with kernels. It also helps to increase parsing speed as further parsing of the recipe is skipped as if it found the current machine is not compatible.
CONFIG_SITE	A list of files which contains autoconf test results relevant to the current build. This variable is used by the autotools utilities when running configure.

D

D	Destination directory
DEBUG_BUILD	Build packages with debugging information. This influences the value SELECTED_OPTIMIZATION takes.
DEBUG_OPTIMIZATION	The options to pass in TARGET_CFLAGS and CFLAGS when compiling a system for debugging. This defaults to "-O -fno-omit-frame-pointer -g".
DEFAULT_PREFERENCE	Priority of recipe
DEPENDS	A list of build time dependencies for a given recipe. These indicate recipes that must have staged before this recipe can configure.
DESCRIPTION	Package description used by package managers
DESTDIR	Destination directory
DISTRO	Short name of distribution
DISTRO_EXTRA_RDEPENDS	List of packages required by distribution.
DISTRO_EXTRA_RRECOMMENDS	List of packages which extend usability of image. Those packages will be automatically installed but can be removed by user.
DISTRO_FEATURES	Features of the distribution.
DISTRO_NAME	Long name of distribution
DISTRO_PN_ALIAS	Alias names of the recipe in various Linux distributions. More information in Configuring the DISTRO_PN_ALIAS variable section
DISTRO_VERSION	Version of distribution
DL_DIR	Directory where all fetched sources will be stored

E

ENABLE_BINARY_LOCALE_GENERATION	Variable which control which locales for glibc are to be generated during build (useful if target device has 64M RAM or less)
EXTRA_OECMAKE	Additional cmake options
EXTRA_OECONF	Additional 'configure' script options
EXTRA_OEMAKE	Additional GNU make options

F

FILES	list of directories/files which will be placed in packages
FULL_OPTIMIZATION	The options to pass in TARGET_CFLAGS and CFLAGS when compiling an optimised system. This defaults to "-fexpensive-optimizations -fomit-frame-pointer -frename-registers -O2".

H

HOMEPAGE	Website where more info about package can be found

I

IMAGE_FEATURES — List of features present in resulting images

IMAGE_FSTYPES — Formats of rootfs images which we want to have created

IMAGE_INSTALL — List of packages used to build image

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

The following example shows how to use INC_PR given a common `.inc` that defines the variable. Once defined, the variable can be used 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 — This variable causes the build to not strip binaries in resulting packages.

INHERIT — This variable causes the named class to be inherited at this point during parsing. Its only valid in configuration files.

INITSCRIPT_PACKAGES — Scope: Used in recipes when using update-rc.d.bbclass. Optional, defaults to PN.

A list of the packages which contain initscripts. If multiple packages are specified you need to append the package name to the other INITSCRIPT_* as an override.

INITSCRIPT_NAME — Scope: Used in recipes when using update-rc.d.bbclass. Mandatory.

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

INITSCRIPT_PARAMS — Scope: Used in recipes when using update-rc.d.bbclass. Mandatory.

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 it in levels 0, 1 and 6.

K

KERNEL_IMAGETYPE — The type of kernel to build for a device, usually set by the machine configuration files and defaults to "zImage". This is used when building the kernel and is passed to "make" as the target to build.

L

LAYERDIR — When used inside a layer.conf gives 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.

LICENSE — 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 the build failure, which gives developer an opportunity to review any license change This is an optional variable now, and the plan is to make it a required variable in the future See "meta/package/zlib/zlib_${PV}.bb" file for an example More information in Configuring the LIC_FILES_CHKSUM variable section

M

MACHINE	Target device
MACHINE_ESSENTIAL_RDEPENDS	List of packages required to boot device
MACHINE_ESSENTIAL_RRECOMMENDS	List of packages required to boot device (usually additional kernel modules)
MACHINE_EXTRA_RDEPENDS	List of packages required to use device
MACHINE_EXTRA_RRECOMMNEDS	List of packages useful to use device (for example additional kernel modules)
MACHINE_FEATURES	List of device features - defined in machine features section
MAINTAINER	E-mail of distribution maintainer

P

PACKAGE_ARCH	Architecture of resulting package
PACKAGE_CLASSES	List of resulting packages formats
PACKAGE_DESCRIPTION	Long form description of binary package for packaging sytems such as ipkg, rpm or debian, inherits DESCRIPTION by default
PACKAGE_EXTRA_ARCHS	List of architectures compatible with device CPU. Usable when build is done for few different devices with misc processors (like XScale and ARM926-EJS)
PACKAGE_SUMMARY	Short (72 char suggested) Summary of binary package for packaging sytems such as ipkg, rpm or debian, inherits DESCRIPTION by default
PACKAGES	List of packages to be created from recipe. The default value is "${PN}-dbg ${PN} ${PN}-doc ${PN}-dev"
PARALLEL_MAKE	Extra options that are passed to the make command during the compile tasks. This is usually of the form '-j 4' where the number represents the maximum number of parallel threads make can run.
PN	Name of package.
PR	Revision of package. The default value is "r0".
PV	Version of package. This is normally extracted from the recipe name, e.g. 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 (git, svn, etc.).
PE	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 one should be given preference. It should be set to the "$PN" of the recipe to be preferred.
PREFERRED_VERSION	If there are multiple versions of recipe available, this variable determines which one should be given preference. It should be set to the "$PV" of the recipe to be preferred.
POKY_EXTRA_INSTALL	List of packages to be added to the image. This should only be set in `local.conf`.
POKYLIBC	Libc implementation selector - glibc, eglibc, or uclibc can be selected.
POKYMODE	Toolchain selector. It can be external toolchain built from Poky or few supported combinations of upstream GCC or CodeSourcery Labs toolchain.

R

RCONFLICTS	List of packages which conflict with this one. Package will not be installed if they are not removed first.
RDEPENDS	A list of run-time dependencies for a package. These packages need to be installed alongside the package it applies to so the package will run correctly, an example is a perl script which would rdepend on perl. Since this variable applies to output packages there would usually be an override attached to this variable like RDEPENDS_ ${PN}-dev. Names in this field should be as they are in PACKAGES namespace before any renaming of the output package by classes like debian.bbclass.
ROOT_FLASH_SIZE	Size of rootfs in megabytes
RRECOMMENDS	List of packages which extend usability of the package. Those packages will be automatically installed but can be removed by user.
RREPLACES	List of packages which are replaced with this one.

S

S	Path to unpacked sources (by default: "${WORKDIR}/${PN}-${PV}")
SECTION	Section where package should be put - used by package managers
SELECTED_OPTIMIZATION	The variable takes the value of FULL_OPTIMIZATION unless DEBUG_BUILD = "1" in which case DEBUG_OPTIMIZATION is used.
SERIAL_CONSOLE	Speed and device for serial port used to attach serial console. This is given to kernel as "console" param and after boot getty is started on that port so remote login is possible.
SHELLCMDS	A list of commands to run within the a shell, used by TERMCMDRUN.
SITEINFO_ENDIANESS	Contains "le" for little-endian or "be" for big-endian depending on the endian byte order of the target system.
SITEINFO_BITS	Contains "32" or "64" depending on the number of bits for the CPU of the target system.
SRC_URI	List of source files (local or remote ones)
SRC_URI_OVERRIDES_PACKAGE_ARCH	By default there is code which automatically detects whether SRC_URI contains files which are machine specific and if this is the case it automatically changes PACKAGE_ARCH. Setting this variable to "0" disables that behaviour.

SRCDATE	Date of source code used to build package (if it was fetched from SCM).
SRCREV	Revision of source code used to build package (Subversion, GIT, Bazaar only).
STAGING_KERNEL_DIR	Directory with kernel headers required to build out-of-tree modules.
STAMPS	Directory (usually TMPDIR/stamps) with timestamps of executed tasks.
SUMMARY	Short (72 char suggested) Summary of binary package for packaging systems such as ipkg, rpm or debian, inherits DESCRIPTION by default

T

TARGET_ARCH	The architecture of the device we're building for. A number of values are possible but Poky primarily supports "arm" and "i586".
TARGET_CFLAGS	Flags passed to C compiler for the target system. Evaluates to the same as CFLAGS.
TARGET_FPU	Method of handling FPU code. For FPU-less targets (most of ARM cpus) it has to be set to "soft" otherwise kernel emulation will get used which will result in performance penalty.
TARGET_OS	Type of target operating system. Can be "linux" for glibc based system, "linux-uclibc" for uClibc. For ARM/EABI targets there are also "linux-gnueabi" and "linux-uclibc-gnueabi" values possible.
TERMCMD	This command is used by bitbake to lauch a terminal window with a shell. The shell is unspecified so the user's default shell is used. By default it is set to gnome-terminal but it can be any X11 terminal application or terminal multiplexers like screen.
TERMCMDRUN	This command is similar to TERMCMD however instead of the users shell it runs the command specified by the SHELLCMDS variable.

W

WORKDIR	Path to directory in tmp/work/ where package will be built.

Appendix G. Reference: Variable Locality (Distro, Machine, Recipe etc.)

Whilst most variables can be used in almost any context (.conf, .bbclass, .inc or .bb file), variables are often associated with a particular locality/context. This section describes some common associations.

G.1. Distro Configuration

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

G.2. Machine Configuration

- TARGET_ARCH
- SERIAL_CONSOLE
- PACKAGE_EXTRA_ARCHS
- IMAGE_FSTYPES
- ROOT_FLASH_SIZE
- MACHINE_FEATURES
- MACHINE_EXTRA_RDEPENDS
- MACHINE_EXTRA_RRECOMMENDS
- MACHINE_ESSENTIAL_RDEPENDS
- MACHINE_ESSENTIAL_RRECOMMENDS

G.3. Local Configuration (local.conf)

- DISTRO
- MACHINE
- DL_DIR
- BBFILES
- IMAGE_FEATURES

- PACKAGE_CLASSES
- BB_NUMBER_THREADS
- BBINCLUDELOGS
- ENABLE_BINARY_LOCALE_GENERATION

G.4. Recipe Variables - Required

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

G.5. Recipe Variables - Dependencies

- DEPENDS
- RDEPENDS
- RRECOMMENDS
- RCONFLICTS
- RREPLACES

G.6. Recipe Variables - Paths

- WORKDIR
- S
- FILES

G.7. Recipe Variables - Extra Build Information

- 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 a derivative of OpenEmbedded [http://www.openembedded.org/], a stable, smaller subset focused on the GNOME Mobile environment. Development in 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. Can I still use Poky?

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:

- http://autobuilder.yoctoproject.org/downloads/miscsupport/python-nativesdk-standalone-i586.tar.bz2
- http://autobuilder.yoctoproject.org/downloads/miscsupport/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;

- We keep Poky small and focused - around 650 packages compared to over 5000 for full OE
- We only support hardware that we have access to for testing
- We have an autobuilder which provides continuous build and integration tests

H.4. How do I get support for my board added to Poky?

There are two main ways to get a board supported in Poky;

- Send us the board if we don't have it yet
- Send us BitBake recipes if you have them (see the Poky handbook to find out how to create recipes)

Usually if it's not a completely exotic board then adding support in Poky should be fairly straightforward.

H.5. Are there any products running poky ?

The Vernier Labquest [http://vernier.com/labquest/] is using Poky (for more about the Labquest see the case study at OpenedHand). There are a number of pre-production devices using Poky and we will announce those as soon as they are released.

H.6. What is the Poky output ?

The output of a Poky build will depend on how it was started, as the same set of recipes can be used to output various formats. Usually the output is a flashable image ready for the target device.

H.7. How do I add my package to Poky?

To add a package you need to create a BitBake recipe - see the Poky handbook to find out how to create a recipe.

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

Poky can build packages in various formats, 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 like Ubuntu or Fedora.

H.9. What is GNOME Mobile? What's 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's wrong?

You're probably running the build on an NTFS filesystem. Use a sane one like ext2/3/4 instead!

H.11. How do I make Poky work in RHEL/CentOS?

To get Poky 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 those the following external packages are needed:

- 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 Poky will be able to build standard images however there may be a problem with QEMU segfaulting. You can either disable the generation of binary locales by setting ENABLE_BINARY_LOCALE_GENERATION to "0" or remove the linux-2.6-execshield.patch from the kernel and rebuild it since its that patch which causes the problems with QEMU.

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

Nothing is wrong, Poky will check any configured source mirrors before downloading from the upstream sources. It does this searching for both source archives and pre-checked out versions of SCM managed software. This is so in large installations, it can reduce load on the SCM servers themselves. The address above is one of the default mirrors configured into standard Poky so if an upstream source disappears, we can place sources there so builds continue to work.

H.13. I have a 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 stop it?

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 which 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 Poky 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/' and 'ftp_proxy = http://proxy.yoyodyne.com:18023/'. Poky also includes a site.conf.sample file which shows how to configure cvs and git proxy servers if needed.

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

In Intrepid, Ubuntu turned on by default 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 " -Wno-format-security -U_FORTIFY_SOURCE" to the BUILD_CPPFLAGS variable in conf/bitbake.conf.

H.16. Whats the difference between foo and foo-native?

The *-native targets are designed to run on the system the build is running on. 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 would run 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 it on has some problem or if you are running it under virtualisation, the virtualisation probably has bugs. Poky processes a massive amount of data causing lots of network, disk and cpu activity and is sensitive to even single bit failure in any of these areas. Totally 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. Its 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 as you are shipping. This means sharing the source code, any patches applied to it but 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 4.1.6, “Miscellaneous Recipe Files” and set the HAVE_TOUCHSCREEN variable equal to one.

```
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 4.1.6, “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. How does Poky obtain source code and will it work behind my firewall or proxy server?

The way Poky obtains source code is highly configurable. You can setup Poky to get source code in most environmnents if HTTP transport is available.

When Poky 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://autobuilder.yoctoproject.org/sources/ \n \
ftp://.*/.* http://autobuilder.yoctoproject.org/sources/ \n \
http://.*/.* http://autobuilder.yoctoproject.org/sources/ \n \
https://.*/.* http://autobuilder.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://autobuilder.yoctoproject.org/sources/ \n \
http://.*/.* http://autobuilder.yoctoproject.org/sources/ \n \
https://.*/.* http://autobuilder.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 Poky

I.1. Introduction

We're happy for people to experiment with Poky and there are a number of places to find help if you run into difficulties or find bugs. To find out how to download source code see the Obtaining Poky section of the Introduction.

I.2. Bugtracker

Problems with Poky should be reported using the Bugzilla application at http://bugzilla.yoctoproject.org/.

I.3. Mailing lists

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

- http://lists.yoctoproject.org/listinfo/yocto for a Yocto Discussions mailing list.
- http://lists.yoctoproject.org/listinfo/poky for a Poky Discussions mailing list.
- http://lists.yoctoproject.org/listinfo/yocto-announce for a mailing list to receive offical Yocto Project announcements for developments and milestones.

I.4. Internet Relay Chat (IRC)

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

- #yocto
- #poky

I.5. Links

- The Yocto Project website [http://yoctoproject.org] - The home site for Yocto Project.
- The Poky website [http://pokylinux.org] - The home site for Poky Linux.
- OpenedHand [http://www.openedhand.com/] - The original company behind Poky.
- Intel Corporation [http://www.intel.com/] - The company who acquired OpenedHand in 2008.
- OpenEmbedded [http://www.openembedded.org/] - The upstream generic embedded distribution Poky derives from (and contributes to).
- Bitbake [http://developer.berlios.de/projects/bitbake/] - The tool used to process Poky 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://fabrice.bellard.free.fr/qemu/] - An open source machine emulator and virtualizer.

I.6. Contributions

Contributions to Poky are very welcome. Patches should be sent to the Poky mailing list along with a Signed-off-by: line in the same style as the Linux kernel. Adding this line signifies the developer has agreed to the Developer's Certificate of Origin 1.1:

```
Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the best
    of my knowledge, is covered under an appropriate open source
    license and I have the right under that license to submit that
    work with modifications, whether created in whole or in part
    by me, under the same open source license (unless I am
    permitted to submit under a different license), as indicated
    in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

A Poky contributions tree (poky-contrib, git://git.yoctoproject.org/poky-contrib.git) exists for people to stage contributions in, for regular contributors. If people desire such access, please ask on the mailing list. Usually access will be given to anyone with a proven track record of good patches.