Poky Reference Manual

Richard PurdieIntel Corporation<richard@linux.intel.com>

This version of the project is now considered obsolete, please select and use a more recent version.

A Guide and Reference to Poky

Tomas Frydrych

Intel Corporation

Marcin Juszkiewicz

Dodji Seketeli

Revision History
Revision 4.0+git 27 Oct 2010
Poky Master Documentation

Chapter 1. Introduction

1.1. Welcome to Poky!

Poky is the the build tool in Yocto Project. It is at the heart of Yocto Project. You use Poky within Yocto Project to build the images (kernel software) for targeted hardware.

Before jumping into Poky you should have an understanding of Yokto Project. Be sure you are familiar with the information in the Yocto Project Quick Start. You can find this documentation on the public Yocto Project Website.

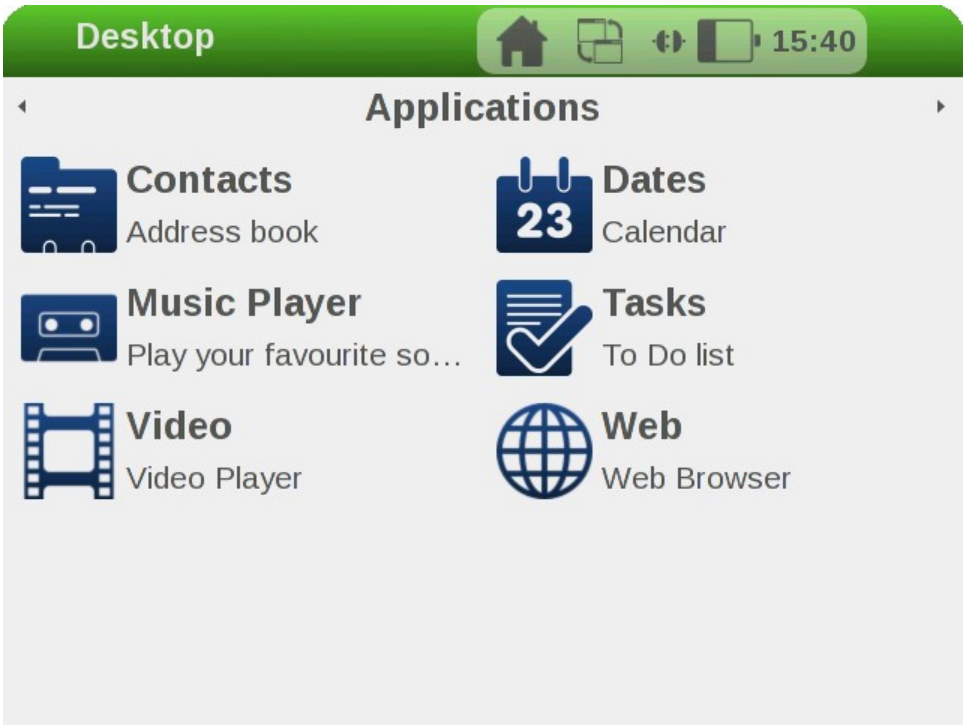
1.2. What is Poky?

Poky provides an open source Linux, X11, Matchbox, GTK+, Pimlico, Clutter, and other GNOME Mobile technologies based full platform build tool within Yocto Project. It creates a focused, stable, subset of OpenEmbedded that can be easily and reliably built and developed upon. Poky fully supports a wide range of x86 ARM, MIPS and PowerPC hardware and device virtulisation.

Poky is primarily a platform builder which 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. Images for many kinds of devices can be generated, however 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 development of 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 at very high DPI and restricted size, such as those often found on smartphones and PDAs. It is coded with focus on efficiency and speed so that it works smoothly on hand-held and other embedded hardware. It will sit neatly on top of any device using the GNOME Mobile stack, providing 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 organisations including Intel Corporation.

1.3. Documentation Overview

The Poky User Guide is split into sections covering different aspects of Poky. The 'Using Poky' section gives an overview of the components that make up Poky followed by information about using Poky and debugging images created in Yocto Project. The 'Extending Poky' section gives information about how to extend and customise Poky along with advice on how to manage these changes. The 'Platform Development with Poky' section gives 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 giving details on a specific section of Poky functionality.

This manual applies to Poky Release 3.3 (Green).

1.4. System Requirements

We recommend Debian-based distributions, in particular a recent Ubuntu release (10.04 or newer), as the host system for Poky. Nothing in Poky is distribution specific and other distributions will

most likely work as long as the appropriate prerequisites are installed - 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 public Yocto Project Website.

1.5. Obtaining Poky

1.5.1. Releases

Periodically, we make releases of Poky and these are available at http://pokylinux.org/releases/. These are more stable and 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.pokylinux.org/ where the numbers increase for each subsequent build and can be used to reference it.

Automated builds are available for "standard" Poky and for Poky SDKs and toolchains as well as any testing versions we might have such as poky-bleeding. The toolchains can be used either as external standalone toolchains or can be combined with Poky as a prebuilt 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.pokylinux.org/poky.git; a web interface to the repository can be accessed at http://git.pokylinux.org/.

The 'master' is where the deveopment work takes place and you should use this if you're after to work with the latest cutting edge developments. It is possible trunk can suffer temporary periods of instability while new features are developed and if this is undesireable we recommend using one of the release branches.

Chapter 2. Using Poky

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

2.1. Poky Overview

At the core of Poky is the bitbake task executor together with various types of configuration files. This section gives an overview of bitbake and the configuration files, in particular what they are used for, and how they interact.

Bitbake handles the parsing and execution of the data files. The data itself is of various types; recipes which give details about particular pieces of software, class data which is an abstraction of common build information (e.g. how to build a Linux kernel) and configuration data for machines, policy decisions, etc., which acts as a glue and binds everything together. Bitbake knows how to combine multiple data sources together, each data source being referred to as a 'layer'.

The directory structure walkthrough section gives details on the meaning of specific directories but some brief details on the core components follows:

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 it supports look at **bitbake --help**.

The most common usage is **bitbake packagename** where packagename is the name of the package you wish to build (from now on called the target). This often equates to the first part of a .bb filename, so to run the `matchbox-desktop_1.2.3.bb` file, you might type **bitbake matchbox-desktop**. Several different versions of matchbox-desktop might exist and bitbake will choose the one selected by the distribution configuration (more details about how bitbake chooses between different versions and providers is available in the 'Preferences and Providers' section). Bitbake will also try to execute any dependent tasks first so before building matchbox-desktop it would build a cross compiler and glibc if not already 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 where to download the source, any patches that are needed, any special configuration options, how to compile the source files and how to package the compiled output.

'package' can also be used to describe recipes but since the same word is used for the packaged output from Poky (i.e. .ipk or .deb files), this document will avoid it.

2.1.3. Classes

Class (.bbclass) files contain information which is useful to share between metadata files. An example is the autotools class which contains the common settings that any application using autotools would use. The classes reference section gives details on common classes and how to use them.

2.1.4. Configuration

The configuration (.conf) files define various configuration variables which govern what Poky does. These are split into several areas, such as machine configuration options, distribution configuration options, compiler tuning options, general common configuration and user configuration (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 building object files. The default build dir is poky-dir/build. Multiple build_dir can be used for different targets. For example, ~/build/x86 for qemux86 target, and ~/build/arm for qemuarm target. Please refer to poky-init-build-env for detail info

Once the Poky build environment is set up, a target can now be built using:

```
$ bitbake <target>
```

The target is the name of the recipe you want to build. Common targets are the images (in `meta/packages/images/`) or the name of a recipe for a specific piece of software like busybox. More details about the standard images are available in the image reference section.

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 covered in the Running an Image section. See your board/machine documentation for information about how to install these images.

2.4. Debugging Build Failures

The exact method for debugging Poky depends on the nature of the bug(s) and which part of the system they might be from. Standard debugging practises such as comparing to the last known working version and examining the changes, reapplying the changes in steps to identify the one causing the problem etc. are valid for Poky just like any other system. It's impossible to detail every possible potential failure here but there are some general tips to aid debugging:

2.4.1. Task Failures

The log file for shell tasks is available in `${WORKDIR}/temp/log.do_taskname.pid`. For the compile task of busybox 1.01 on the ARM spitz machine, this might be `tmp/work/armv5te-poky-linux-gnueabi/busybox-1.01/temp/log.do_compile.1234` for example. To see what bitbake ran to generate that log, look at the `run.do_taskname.pid` file in the same directory.

The output from python tasks is sent directly to the console at present.

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 this depends on are built first hence the standard bitbake behaviour. There are some tasks such as devshell which are not part of the default build chain. If you wish to run such a task you can use the "-c" option to bitbake e.g. **bitbake matchbox-desktop -c devshell**.

If you wish to rerun a task you can use the force option "-f". A typical usage session might look like:

```
% bitbake matchbox-desktop
[change some source in the WORKDIR for example]
% bitbake matchbox-desktop -c compile -f
% bitbake matchbox-desktop
```

which would build matchbox-desktop, then recompile it. The final command reruns all tasks after the compile (basically the packaging tasks) since bitbake will notice that the compile has been rerun and hence the other tasks also need to run again.

You can view a list of tasks in a given package by running the listtasks task e.g. **bitbake**

matchbox-desktop -c listtasks, and the result is in 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. **bitbake -g targetname** will create `depends.dot` and `task-depends.dot` files in the current directory. They show which packages and tasks depend on which other packages and tasks and are useful for debugging purposes. **"bitbake -g -u depexp targetname"** will show result in more human-readable GUI style.

2.4.4. General Bitbake Problems

Debug output from bitbake can be seen with the "-D" option. The debug output gives more information about what bitbake is doing and/or why. Each -D option increases the logging level, the most common usage being "-DDD".

The output from **bitbake -DDD -v targetname** can reveal why a certain version of a package might be chosen, why bitbake picked a certain provider or help in other situations where bitbake does something you're not expecting.

2.4.5. Building with no dependencies

If you really want to build a specific .bb file, you can use the form **bitbake -b somepath/somefile.bb**. Note that this will not check the dependencies so this option should only be used when you know its dependencies already exist. You can specify fragments of the filename and bitbake will see if it can find a unique match.

2.4.6. Variables

The "-e" option will dump 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 keeping an eye open for bad things creeping into compiler commandlines such as references to local system files (`/usr/lib/` or `/usr/include/` etc.).

Tip

If you want to remove the psplash boot splashscreen, add "psplash=false" to the kernel commandline and psplash won't load allowing you to see the console. It's also possible to switch out of the splashscreen by switching virtual console (Fn+Left or Fn+Right on a Zaurus).

Chapter 3. Extending Poky

This section gives information about how to extend the functionality already present in Poky, documenting standard tasks such as adding new software packages, extending or customising images or porting poky to new hardware (adding a new machine). It also contains advice about how

to manage the process of making changes to Poky to achieve best results.

3.1. Adding a Package

To add package into Poky you need to write a recipe for it. Writing a recipe means creating a .bb file which sets various variables. The variables useful for recipes are detailed in the recipe reference section along with more detailed information about issues such as recipe naming.

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 hence a wider range of packages. Poky aims to be compatible with OpenEmbedded so most recipes should just work in Poky.

For new packages, the simplest way to add a recipe is to base it on a similar pre-existing recipe. There are some examples below of how to add standard types of packages:

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

To build an application from a single file stored locally (e.g. under "files/") requires a recipe which has the file listed in the *SRC_URI* variable. In addition the `do_compile` and `do_install` tasks need to be manually written. The *S* variable defines the directory containing the source code which in this case is set equal to *WORKDIR*, 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}
}
```

As a result of the build process "helloworld", "helloworld-dbg" and "hellworld-dev" packages will be built by default. It is possible to customise the packaging process.

3.1.2. Autotooled Package

Applications which use autotools (autoconf, automake) require a recipe which has a source archive listed in *SRC_URI* and **inherit autotools** to instruct BitBake to use the `autotools.bbclass` which has definitions of all the steps needed to build an autotooled application. The result of the build will be automatically packaged and if the application uses NLS to localise then packages with locale information will be generated (one package per language). Below 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
```

LIC_FILES_CHKSUM is used to track source license change. Autotool based recipe can be quickly created this way like above example.

3.1.3. Makefile-Based Package

Applications which use GNU make require a recipe which has the source archive listed in *SRC_URI*. Adding a `do_compile` step is not needed as by default BitBake will start the "make" command to compile the application. If there is a need for additional options to make then they should be stored in the *EXTRA_OEMAKE* variable - BitBake will pass them into the GNU make invocation. A `do_install` task is required - otherwise BitBake will run an empty `do_install` task by default.

Some applications may require extra parameters to be passed to the compiler, for example an additional header path. This can be done buy adding to the *CFLAGS* variable, as in the example below:

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

mtd-utils is an example as Makefile-based:

```
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 packages content

The variables *PACKAGES* and *FILES* are used to split an application into multiple packages.

Below the "libXpm" recipe (libxpm_3.5.7.bb) is used as an example. By default the "libXpm" recipe

generates one package which contains the library and also a few binaries. The recipe can be adapted 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 this example we want to ship the "sxpm" and "cxpm" binaries in separate packages. Since "bindir" would be packaged into the main *PN* package as standard we prepend the *PACKAGES* variable so additional package names are added to the start of list. The extra *FILES*_* variables then contain information to specify which files and directories goes into which package. Files included by earlier package are skipped by latter packages, and thus main *PN* will not include 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 where PACKAGENAME is the name of the package to attach the postinst script to. Normally *PN* can be used which expands to PACKAGENAME automatically. 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 gets 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 will be executed again on the first boot of the image.

Sometimes it is necessary that the execution of a post-installation script is delayed until the first boot, because the script needs to be executed on the device itself. To delay script execution until boot time, the post-installation function should have the following structure:

```
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 structure above delays execution until first boot 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. Customising Images

Poky images can be customised to satisfy particular requirements. Several methods are detailed below along with guidelines of when to use them.

3.2.1. Customising Images through a custom image .bb files

One way to get additional software into an image is by creating a custom image. The recipe will contain two lines:

```
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. The names must be in the OpenEmbedded notation instead of Debian notation, for example "glibc-dev" instead of "libc6-dev" etc.

The other method of creating a new image is by modifying an existing image. For example if a developer wants to add "strace" into "poky-image-sato" the following recipe can be used:

```
require poky-image-sato.bb

IMAGE_INSTALL += "strace"
```

3.2.2. Customising Images through custom tasks

For complex custom images, the best approach is to create a custom task package which is then used to build the image (or images). A good example of a tasks package is `meta/packages/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, *RDEPENDS* and *RRECOMMENDS* entries can then be added each containing a list of packages the parent task package should contain. An example would be:

```
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 this example, two task packages are created, task-custom-apps and task-custom-tools with the dependencies and recommended package dependencies listed. To build an image using these task packages, you would then add "task-custom-apps" and/or "task-custom-tools" to *IMAGE_INSTALL* or other forms of image dependencies as described in other areas of this section.

3.2.3. Customising Images through custom *IMAGE_FEATURES*

Ultimately users may want to add extra image "features" as used by Poky with the *IMAGE_FEATURES* variable. To create these, the best reference is `meta/classes/poky-image.bbclass` which illustrates how poky achieves this. In summary, the file looks at the contents of the *IMAGE_FEATURES* variable and then maps this into a set of tasks or packages. Based on this then the *IMAGE_INSTALL* variable is generated automatically. Extra features can be added by extending the class or creating a custom class for use with specialised image .bb files.

3.2.4. Customising Images through local.conf

It is possible to customise image contents by abusing variables used by distribution maintainers in local.conf. This method only allows the addition of packages and is not recommended.

To add an "strace" package into the image the following is added to local.conf:

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

However, since the *DISTRO_EXTRA_RDEPENDS* variable is for distribution maintainers this method does not make adding packages as simple as a custom .bb file. Using this method, a few packages will need to be recreated if they have been created before and then the image is rebuilt.

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

Cleaning task-* packages is required because they use the *DISTRO_EXTRA_RDEPENDS* variable. There is no need to build them by hand as Poky images depend on the packages they contain so dependencies will be built automatically when building the image. For this reason we don't use the "rebuild" task in this case since "rebuild" does not care about dependencies - it only rebuilds the specified package.

3.3. Porting Poky to a new machine

Adding a new machine to Poky is a straightforward process and this section gives an idea of the changes that are needed. This guide is meant to cover adding machines similar to those Poky already supports. Adding a totally new architecture might require gcc/glibc changes as well as updates to the site information and, whilst well within Poky's capabilities, is outside the scope of this section.

3.3.1. Adding the machine configuration file

A .conf file needs to be added to conf/machine/ with details of the device being added. The name of the file determines the name Poky will use to reference this 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"). Other variables like *SERIAL_CONSOLE* (e.g. "115200 ttyS0"), *KERNEL_IMAGETYPE* (e.g. "zImage") and *IMAGE_FSTYPES* (e.g. "tar.gz jffs2") might also be needed. Full details on what these variables do and the meaning of their contents is available through the links. There're lots of existing machine .conf files which can be easily leveraged 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. There are plenty of kernel examples in the meta/recipes-kernel/linux directory which can be used as references.

If creating a new recipe the "normal" recipe writing rules apply for setting up a *SRC_URI* including any patches and setting *S* to point at the source code. You will need to create a configure task which configures the unpacked kernel with a defconfig be that through a "make defconfig" command or more usually though copying in a suitable defconfig and running "make oldconfig". By making use of "inherit kernel" and also maybe some of the linux-*.inc files, most other functionality is centralised and the the defaults of the class normally work well.

If extending an existing kernel it is usually a case of adding a suitable defconfig file in a location similar to that used by other machine's defconfig files in a given kernel, possibly listing it 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 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 screen resolution.

Sane defaults should be used in most cases, but if customisation is necessary you need to create a `machconfig` file under `meta/packages/formfactor/files/MACHINENAME/` where `MACHINENAME` is the name for which this infomation applies. For information about the settings available and the defaults, please see `meta/packages/formfactor/files/config`. Below is one 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

We recognise that people will want to extend/configure/optimise Poky for their specific uses, especially due to the extreme configurability and flexibility Poky offers. To ensure ease of keeping pace with future changes in Poky we recommend making changes to Poky in a controlled way.

Poky supports the idea of "layers" which when used properly can massively ease future upgrades and allow segregation between the Poky core and a given developer's changes. Some other advice on managing changes to Poky is also given in the following section.

3.4.1. Bitbake Layers

Often, people want to extend Poky either through adding packages or 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 which demonstrate this functionality, such as meta-emenlow and meta-extras. The meta-emenlow layer is an example layer enabled by default. The meta-extras repostory is not enabled by default but enabling any layer is as easy as adding the layers path to the BBLAYERS variable in your bblayers.conf. this is how meta-extras are enabled in Poky builds:

```
LCONF_VERSION = "1"

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

Bitbake parses the conf/layer.conf of each of the layers in BBLAYERS to add the recipes, classes and configuration contained within the layer to Poky. To create your own layer, independent of the main Poky repository, you need only create a directory with a conf/layer.conf file and add the directory to your bblayers.conf.

The meta-emenlow/conf/layer.conf 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"
```

As can be seen, the layers recipes are added to *BBFILES*. The BBFILE_COLLECTIONS variable is then appended to with the layer name. The BBFILE_PATTERN variable is immediately expanded

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. This is useful in situations where the same package might appear in multiple layers and allows you to choose which layer should 'win'. Note the use of *LAYERDIR* with the immediate expansion operator. *LAYERDIR* expands to the directory of the current layer and requires use of the immediate expansion operator so that Bitbake does not lazily expand the variable when it's parsing a different directory.

Additional bbclass and configuration files can be locationed by bitbake through the addition to the BBPATH environment variable. In this case, the first file with the matching name found in BBPATH is the one that is used, just like the PATH variable for binaries. It is therefore recommended that you use unique bbclass and configuration file names in your custom layer.

The recommended approach for custom layers is to store them in a git repository of the format meta-prvt-XXXX and have this repository cloned alongside the other meta directories in the Poky tree. This way you can keep your Poky tree and it's configuration entirely inside POKYBASE.

3.4.2. Committing Changes

Modifications to Poky are often managed under some kind of source revision control system. The policy for committing to such systems is important as some simple policy can significantly improve usability. The tips below are based on the policy followed for the Poky core.

It helps to use a consistent style for commit messages when committing changes. We've found a style where the first line of a commit message summarises the change and starts with the name of any package affected work well. Not all changes are to specific packages so the prefix could also be a machine name or class name instead. If a change needs a longer description this should follow the summary:

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

These functions allow generation of dependency data between funcitons 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 rpurdie@linux.intel.com
```

Any commit should be self contained in that it should leave the metadata in a consistent state, buildable before and after the commit. This helps ensure the autobuilder test results are valid but is good practice regardless.

3.4.3. Package Revision Incrementing

If a committed change will result in changing the package output then the value of the *PR* variable needs to be increased (commonly referred to as 'bumped') as part of that commit. Only integer values are used and *PR* = "r0" should be added into new recipes as, while this is the default value, not having the variable defined in a recipe makes it easy to miss incrementing it when updating the recipe. When upgrading the version of a package (*PV*), the *PR* variable should be reset to "r0".

The aim is that the package version will only ever increase. If for some reason *PV* will change and but not increase, the *PE* (Package Epoch) can be increased (it defaults to '0'). The version numbers aim to follow the Debian Version Field Policy Guidelines which define how versions are compared and hence what "increasing" means.

There are two reasons for doing this, the first is 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. The second is to ensure that target users are able to upgrade their devices via their package manager such as with the **opkg upgrade** commands (or similar for dpkg/apt or rpm based systems). The aim is to ensure Poky has upgradable packages in all cases.

3.4.4. Using Poky in a Team Environment

It may not be immediately clear how Poky can work in a team environment, or scale to a large team of developers. The specifics of any situation will determine the best solution and poky offers immense flexibility in that aspect but there are some practises that experience has shown to work well.

The core component of any development effort with Poky is often an automated build testing framework and image generation process. This can be used to check that the metadata is buildable, highlight when commits break the builds and provide up to date images allowing people to test the end result and use them as a base platform for further development. Experience shows that buildbot is a good fit for this role and that it works well to configure it to make two types of build - incremental builds and 'from scratch'/full builds. The incremental builds can be tied to a commit hook which triggers them each time a commit is made to the metadata and are a useful acid test of whether a given commit breaks the build in some serious way. They catch lots of simple errors and whilst they won't catch 100% of failures, the tests are fast so developers can get feedback on their changes quickly. The full builds are builds that build everything from the ground up and test everything. They usually happen at preset times such as at night when the machine load isn't high from the incremental builds. poky autobuilder is an example implementation with buildbot.

Most teams have pieces of software undergoing active development. It is of significant benefit to put these under control of a source control system compatible with Poky such as git or svn. The autobuilder can then be set to pull the latest revisions of these packages so the latest commits get tested by the builds allowing any issues to be highlighted quickly. Poky easily supports configurations where there is both a stable known good revision and a floating revision to test. Poky can also only take changes from specific source control branches giving another way it can be used to track/test only specified changes.

Perhaps the hardest part of setting this up is the policy that surrounds the different source control systems, be them software projects or the Poky metadata itself. The circumstances will be different in each case but this is one of Poky's advantages - the system itself doesn't force any particular policy unlike a lot of build systems, allowing the best policy to be chosen for the circumstances.

3.4.5. Updating Existing Images

Often, rather than reflashing a new image you might wish to install updated packages into an existing running system. This can be done by sharing the `tmp/deploy/ipk/` directory through a web server and then on the device, changing `/etc/opkg/base-feeds.conf` to point at this server, for example by adding:

```
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

Poky is usually used to build software rather than modifying it. However, there are ways Poky can be used to modify software.

During building, the sources are available in *WORKDIR* directory. Where exactly this is depends on the type of package and the architecture of target device. For a standard recipe not related to *MACHINE* it will be `tmp/work/PACKAGE_ARCH-poky-TARGET_OS/PN-PV-PR/`. Target device dependent packages use *MACHINE* instead of *PACKAGE_ARCH* in the directory name.

Tip

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

After building a package, a user can modify the package source code without problem. The easiest way to test changes is by calling the "compile" task:

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

"-f" or "--force" is used to force re-execution of the specified task. Other tasks may also be called this way. But note that all the modifications in *WORKDIR* are gone once you executes "-c clean" for a package.

3.5.1. Modifying Package Source Code with quilt

By default Poky uses quilt to manage patches in `do_patch` task. It is a powerful tool which can be used to track all modifications done to package sources.

Before modifying source code it is important to notify quilt so it will track changes into new patch file:

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

Then add all files which will be modified into that patch:

```
quilt add file1 file2 file3
```

Now start editing. At the end quilt needs to be used to generate final patch which will contain all modifications:

```
quilt refresh
```

The resulting patch file can be found in the `patches/` subdirectory of the source (*S*) directory. For future builds it should be copied into Poky metadata and added into *SRC_URI* of a recipe:

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

This also requires a bump of *PR* value in the same recipe as we changed resulting packages.

3.6. Track license change

The license of one upstream project may change in the future, and Poky provides one mechanism to track such license change - *LIC_FILES_CHKSUM* variable.

3.6.1. Specifying the LIC_FILES_CHKSUM variable

```
LIC_FILES_CHKSUM = "file://COPYING; md5=xxxx \
                    file://licfile1.txt; beginline=5; endline=29;md5=yyyy \
                    file://licfile2.txt; endline=50;md5=zzzz \
                    ..."
```

S is the default directory for searching files listed in *LIC_FILES_CHKSUM*. Relative path could be used too:

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

The first line locates a file in *S*/src/ls.c, and the second line refers to a file in *WORKDIR*, which is the parent of *S*

3.6.2. Explanation of syntax

This parameter lists all the important files containing the text of licenses for the source code. It is also possible to specify on which line the license text starts and on which line it ends within that file using the "beginline" and "endline" parameters. If the "beginline" parameter is not specified then license text begins from the 1st line is assumed. Similarly if "endline" parameter is not specified then the license text ends at the last line in the file is assumed. So if a file contains only licensing information, then there is no need to specify "beginline" and "endline" parameters.

The "md5" parameter stores the md5 checksum of the license text. So if the license text changes in any way from a file, then its md5 sum will differ and will not match with the previously stored md5 checksum. This mismatch will trigger build failure, notifying developer about the license text md5 mismatch, and allowing the developer to review the license text changes. Also note that if md5 checksum is not matched while building, the correct md5 checksum is printed in the build log which can be easily copied to .bb file.

There is no limit on how many files can be specified on this parameter. But generally every project would need specifying of just one or two files for license tracking. Many projects would have a "COPYING" file which will store all the license information for all the source code files. If the "COPYING" file is valid then tracking only that file would be enough.

Tip

1. If you specify empty or invalid "md5" parameter; then while building the package, bitbake will give md5 not matched error, and also show the correct "md5" parameter value both on the screen and in the build log

2. If the whole file contains only license text, then there is no need to specify "beginline" and "endline" parameters.

3.7. Handle package name alias

Poky implements a distro_check task which automatically connects to major distributions and checks whether they contains same package. Sometimes the same package has different names in different distributions, which results in a mismatch from distro_check task This can be solved by defining per distro recipe name alias - *DISTRO_PN_ALIAS*

3.7.1. Specifying the DISTRO_PN_ALIAS variable

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

Use space as the delimiter if there're multiple distro aliases

Tip

The current code can check if the src package for a recipe exists in the latest releases of these distributions automatically.

```
Fedora, OpenSuSE, Debian, Ubuntu, Mandriva
```

For example, this command will generate a report, listing which linux distros include the sources for each of the poky recipe.

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

The results will be 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 which together defines how to support a particular hardware device, set of devices, or hardware platform. It will include information about the hardware features present on the device and kernel configuration information along with any additional hardware drivers required. It will also list any additional software components required in addition to a generic Linux software stack for both essential and optional platform features.

The intent of this document is to define a structure for these components so that BSPs follow a commonly understood layout, allowing them to be provided in a common form that everyone understands. It also allows end-users to become familiar with one common format and encourages standardisation 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/OpenEmbedded and that it will be simple to extract information and convert to other formats if required. The format described can be directly accepted as a layer by Poky using its standard layers mechanism, but it is important to recognise that 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 in use.

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 the BSP may be shipped combined with a build system and other tools, but it is important to maintain the distinction that these are separate components which may just be combined in certain end products.

4.1. Example Filesystem Layout

The BSP consists of a file structure inside a base directory, meta-bsp in this example, where "bsp" is a placeholder for the machine or platform name. Examples of some files that it could contain are:

```
meta-bsp/
meta-bsp/binary/zImage
meta-bsp/binary/poky-image-minimal.directdisk
meta-bsp/conf/layer.conf
meta-bsp/conf/machine/*.conf
meta-bsp/conf/machine/include/tune-*.inc
meta-bsp/packages/bootloader/bootloader_0.1.bb
meta-bsp/packages/linux/linux-bsp-2.6.50/*.patch
meta-bsp/packages/linux/linux-bsp-2.6.50/defconfig-bsp
meta-bsp/packages/linux/linux-bsp_2.6.50.bb
meta-bsp/packages/modem/modem-driver_0.1.bb
meta-bsp/packages/modem/modem-daemon_0.1.bb
meta-bsp/packages/image-creator/image-creator-native_0.1.bb
meta-bsp/prebuilds/
```

The following sections detail what these files and directories could contain.

4.2. Prebuilt User Binaries (meta-bsp/binary/*)

This optional area contains useful prebuilt kernels and userspace filesystem images appropriate to the target system. Users could use these to get a system running and quickly get started on development tasks. The exact types of binaries present will be highly hardware-dependent but a README file should be present explaining how to use them with the target hardware. If prebuilt binaries are present, source code to meet licensing requirements must also be provided in some form.

4.3. Layer Configuration (meta-bsp/conf/layer.conf)

This file identifies the structure as a Poky layer. This file identifies the contents of the layer and contains information about how Poky should use it. In general it will most likely be a standard boilerplate file consisting of:

```
# 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"
```

which simply makes bitbake aware of the recipes and conf directories.

This file is required for recognition of the BSP by Poky.

4.4. Hardware Configuration Options (meta-bsp/conf/machine/*.conf)

The machine files bind together all the information contained elsewhere in the BSP into a format that Poky/OpenEmbedded can understand. If the BSP supports multiple machines, multiple machine configuration files can be present. These filenames correspond to the values users set the MACHINE variable to.

These files would define things like which kernel package to use (PREFERRED_PROVIDER of virtual/kernel), which 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 Poky BSP layer but more than one may be present.

4.5. Hardware Optimisation Options (meta-bsp/conf/machine/include/tune-*.inc)

These are shared hardware "tuning" definitions and are commonly used to pass specific optimisation 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"
```

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

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

These files are optional for a Poky BSP layer.

4.6. Linux Kernel Configuration (meta-bsp/packages/linux/*)

These files make up the definition of a kernel to use with this hardware. In this case it is a complete self-contained kernel with its own configuration and patches but kernels can be shared between many machines as well. Taking some specific example files:

```
meta-bsp/packages/linux/linux-bsp_2.6.50.bb
```

which is the core kernel recipe which firstly details where to get the kernel source from. All standard source code locations are supported so this could be a release tarball, some git repository, or source included in the directory within the BSP itself. It then contains information about which patches to apply and how to configure and build it. It can reuse the main Poky kernel build class, so

the definitions here can remain very simple.

```
linux-bsp-2.6.50/*.patch
```

which are patches which may be applied against the base kernel, wherever they may have been obtained from.

```
meta-bsp/packages/linux/linux-bsp-2.6.50/defconfig-bsp
```

which is the configuration information to use to configure the kernel.

Examples of kernel recipes are available in Poky itself. These files are optional since a kernel from Poky itself could be selected, although it would be unusual not to have a kernel configuration.

4.7. Other Software (meta-bsp/packages/*)

This area includes other pieces of software which the hardware may need for best operation. These are just examples of the kind of things that may be encountered. These are standard .bb file recipes in the usual Poky format, so for examples, see standard Poky recipes. The source can be included directly, referred to in source control systems or release tarballs of external software projects.

```
meta-bsp/packages/bootloader/bootloader_0.1.bb
```

Some kind of bootloader recipe which may be used to generate a new bootloader binary. Sometimes these are included in the final image format and needed to reflash hardware.

```
meta-bsp/packages/modem/modem-driver_0.1.bb
meta-bsp/packages/modem/modem-daemon_0.1.bb
```

These are examples of a hardware driver and also a hardware daemon which may need to be included in images to make the hardware useful. "modem" is one example but there may be other components needed like firmware.

```
meta-bsp/packages/image-creator/image-creator-native_0.1.bb
```

Sometimes the device will need an image in a very specific format for its update mechanism to accept and reflash with it. Recipes to build the tools needed to do this can be included with the BSP.

These files only need be provided if the platform requires them.

4.8. Append BSP specific information to existing recipes

Say you have a recipe like pointercal which has machine-specific information in it, and then you have your new BSP code in a layer. Before the .bbappend extension was introduced, you'd have to copy the whole pointercal recipe and files into your layer, and then add the single file for your machine, which is ugly. .bbappend makes the above work much easier, to allow BSP-specific information to be merged with the original recipe easily. When bitbake finds any X.bbappend files, they will be included after bitbake loads X.bb but before finalise or anonymous methods run. This allows the BSP layer to poke around and do whatever it might want to customise the original recipe.

If your recipe needs to reference extra files it can use the FILESEXTRAPATH variable to specify their location. The example below shows extra files contained in a folder called ${PN} (the package name).

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

Then the BSP could add machine-specific config files in layer directory, which will be added by bitbake. You can look at meta-emenlow/packages/formfactor as an example.

4.9. Prebuild Data (meta-bsp/prebuilds/*)

The location can contain a precompiled representation of the source code contained elsewhere in the BSP layer. It can be processed and used by Poky to provide much faster build times, assuming a compatible configuration is used.

These files are optional.

4.10. BSP 'Click-through' Licensing Procedure

Note

This section is here as a description of how click-through licensing is expected to work, and is not yet not impemented.

In some cases, a BSP may contain separately licensed IP (Intellectual Property) for a component, which imposes upon the user a requirement to accept the terms of a 'click-through' license. Once the license is accepted (in whatever form that may be, see details below) the Poky build system can then build and include the corresponding component in the final BSP image. Some affected components may 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. Other components may be simply 'good-to-have' or purely elective, or if essential nonetheless have a 'free' (possibly less-capable) version which may substituted for in the BSP recipe.

For the latter cases, where it is possible to do so from a functionality perspective, the Poky website will make available a 'de-featured' BSP completely free of encumbered IP, which can be used directly and without any further licensing requirements. If present, this fully 'de-featured' BSP will be named meta-bsp (i.e. the normal default naming convention). This is the simplest and therefore preferred option if available, assuming the resulting functionality meets requirements.

If however, a non-encumbered version is unavailable or the 'free' version would provide unsuitable functionality or quality, an encumbered version can be used. Encumbered versions of a BSP are given names of the form meta-bsp-nonfree. There are several ways within the Poky build system to satisfy the licensing requirements for an encumbered BSP, in roughly the following order of preference:

- Get a license key (or keys) for the encumbered BSP by visiting https://pokylinux.org/bsp-keys.html and give the web form there the name of the BSP and your e-mail address.

  ```
  [screenshot of dialog box]
  ```

  After agreeing to any applicable license terms, the BSP key(s) will be immediately sent to the

address given and can be used by specifying BSPKEY_<keydomain> environment variables when building the image:

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

This will 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 may be multiple licenses in effect for a give BSP; a given <keydomain> in such cases corresponds to a particular license. In order for an encumbered BSP encompassing 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.

- Do nothing - build as you normally would, and follow any license prompts that originate from the encumbered BSP (the build will cleanly stop at this point). These usually take the form of instructions needed to manually fetch the encumbered package(s) and md5 sums into e.g. the poky/build/downloads directory. Once the manual package fetch has been completed, restarting the build will continue where it left off, this time without the prompt since the license requirements will have been satisfied.
- Get a full-featured BSP recipe rather than a key, by visiting https://pokylinux.org/bsps.html. Accepting the license agreement(s) presented will subsequently allow you to download a tarball containing a full-featured BSP legally cleared for your use by the just-given license agreement(s). This method will also allow the encumbered image to be built with no change at all to the normal build process.

Note that method 3 is also the only option available when downloading pre-compiled images generated from non-free BSPs. Those images are likewise available at https://pokylinux.org/bsps.html.

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 (see the images section) build tarballs that contain toolchains and libraries suitable for application development outside of Poky. These tarballs unpack into the `/opt/poky` directory and contain a setup script (e.g. `/opt/poky/environment-setup-i586-poky-linux`, which you can source to initialize a suitable environment. Sourcing these 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 set 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 as in the following 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 supports 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). 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://yocto./download (real link needed).

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.pokylinux.org/cgit.cgi/eclipse-poky.

5.1.2.1.1. Installing and Setting up the Eclipse IDE

If you don't have the Eclipse IDE (Helios 3.6) 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 is used to store 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 permenant 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 configure 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:

- Select the "Help -> Install New Software" item.
- In the "Work with:" area click "Add..." and enter the URL for the Yocto plug-in (we need to supply this URL).
- 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 then the architecture with which you will be working. Start by selecting "Preferences" from the "Window" menu and then selecting "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 will need to provide your poky tree path, for example, $<Poky_tree>/build/.

Now 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 HW. 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, Please refer to QEMU section for how to set it up. (Section TBD)

Save all your settings and they become your defaults for every new Yocto project created using the Eclipse IDE.

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 autotools based project, deploy it into QEMU, and then run the debugger against it. You need to configure the project, trigger **autogen.sh**, build the image, start QEMU, and then debug.

1. Creating a Yocto Autotools Based Project Using a Template: Get to the Wizard selection by selecting the File -> New -> Project menu. Expand "C/C++" and select "C Project". Click "Next" and select a template to start with, for example "Hello World ANSI C Project". Complete the steps to create a new Yocto autotools based project using this template.
2. Specify Specific Toolchain Configurations: 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 Project -> Invoke Yocto Tools -> Reconfigure Yocto. Use this dialogue to specify specific toolchain and QEMU setups for the project.
3. Building the Project: Trigger **autogen.sh** through Project -> Reconfigure Project. Then build the project using Project -> Build.
4. Starting QEMU: Use the Run -> External Tools menu and see if there is a QEMU instance for the desired target. If there is click on the instance to start QEMU. If your target is not there then click "External Tools Configuration". You should find an instance of QEMU for your architecture under the entry under "Program". After the boot completes you are ready to

deploy the image into QEMU.

5. Debugging: To bring up your remote debugging configuration in the right-hand window highlight your project in “Project Explorer”, select the Run -> Debug Configurations menu item and expand “C/C++ Remote Application”. Next, select projectname_ gdb_target-poky-linux. You need to be sure that there is an entry for the remote target you want to deploy and cross debug with. If there is no entry then click "New..." to bring up the wizard. Using the wizard select TCF and enter the IP address of you remote target in the “Host name:” field. Back in the remote debug configure window, you need to specify the absolute path for the program on the remote target in the “Remote Absolute File Path for C/C++ Application” field. By default, the program deploys into the remote target. If you don't want this then check “Skip download to target path”. Finally, click "Debug” to start the remote debugging session.

5.1.2.1.5. Using Yocto Eclipse plug-in Remote Tools Suite

Remote tools let you do things like 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 have to select an existing TCF-based RSE connection to the remote target. If one does not exist you need to create one by clicking "New"

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. To use the oprofile the oprofile-viewer must be installed on the local host machine and the oprofile-server must be installed on the remote target.
- lttng: Selecting this tool runs ustrace on the remote target, transfers the output data back to the local host machine and uses lttv-gui to graphically display the output. To use this tool the lttv-gui must be installed on the local host machine. See for information on how to use **lttng** to trace an application.

  For "Application" you must supply the absolute path name to 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 result 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: The **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 in 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: We will stop Anjuta plug-in support after Yocto project 0.9 release. Its source code can be downloaded from git respository listed below, and free for the community to continue supporting it moving forward.

An Anjuta IDE plugin exists to make developing software within the Poky framework easier for the application developer. It presents a graphical IDE with which you can cross compile an application then deploy and execute the output in a QEMU emulation session. It also supports cross debugging and profiling.

To use the plugin, a toolchain and SDK built by Poky is required, Anjuta, it's development headers and the Anjuta plugin. The Poky Anjuta plugin 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.pokylinux.org/anjuta-poky. You can also access a web interface to the repository at http://git.pokylinux.org/?p=anjuta-poky.git;a=summary.

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

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 uses glade for the UI.
3. To activate the plug-in go to Edit -> Preferences, 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. See the next section to learn how to configure it.

5.1.2.2.2. Configuring the Anjuta Plugin

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

- SDK root: If you use an external toolchain you need to set SDK root. This is the root directory of the SDK's sysroot. For an i586 SDK this will be **/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 comple 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. This 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 name. For example, "i586-poky-linux" is extracted from the set up script file **/opt/poky/environment-setup-i586-poky-linux**.
- Kernel: Use the file chooser to select the kernel to use with QEMU.
- Root filesystem: Use the file chooser to select the root filesystem directory. This directory is

where you use the **poky-extract-sdk** to extract the poky-image-sdk tarball.

5.1.2.2.3. Using the Anjuta Plug-in

This section uses an example that cross-compiles a project, deploys it into QEMU, runs a debugger against it and then does 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. Select 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. Start QEMU by selecting Tools -> Start QEMU. This menu selection starts QEMU and will show any error messages in the message view. Once Poky has fully booted within QEMU you can now deploy the project into it.
4. Once the project is built and you have QEMU running choose Tools -> Deploy. This selection installs the package into a temporary directory and then copies using rsync over SSH into the target. Progress and messages appear in the message view.
5. To debug a program installed onto the target choose Tools -> Debug remote. This selection prompts you for the local binary to debug and also the command line to run on the target. The command line to run should include the full path to the to binary installed in the target. This will start a gdbserver over SSH on the target and also an instance of a cross-gdb in a local terminal. This will be preloaded to connect to the server and use the SDK root to find symbols. This gdb will connect to the target and load in various libraries and the target program. You should setup any breakpoints or watchpoints now since you might not be able to interrupt the execution later. You can stop the debugger on the target using Tools -> Stop debugger.
6. It is also possible to execute a command in the target over SSH, the appropriate environment will be be set for the execution. Choose Tools -> Run remote to do this. This selection opens a terminal with the SSH command inside.
7. To do a system wide profile against the system running in QEMU choose Tools -> Profile remote. This selection starts up OProfileUI with the appropriate parameters to connect to the server running inside QEMU and also supplies the path to the debug information necessary to get a useful profile.

5.1.3. Developing externally in QEMU

Running Poky QEMU images is covered in the Running an Image section.

Poky's QEMU images contain a complete native toolchain. This means that applications can be developed within QEMU in the same was as a normal system. Using qemux86 on an x86 machine is fast since the guest and host architectures match, qemuarm is 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 too. 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 will automatically be used from within qemu simply by calling distcc (**export CC="distcc"** can be set in the enviroment). Alterntatively, if a suitable SDK/toolchain is present in `/opt/poky` it will also automatically be used.

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

dropbear ssh server running with the root password disabled allowing standard ssh and scp commands to work. The images also contain an NFS server exporting the guest's root filesystem allowing that 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
```

Here, we build the package, change into the work directory for the package, change a file, then recompile the package. Instead of using sh like this, you can also use two different terminals. The risk with working like this is that a command like unpack could wipe out the changes you've made to the work directory so 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. The resulting patches can be copied into the recipe directory and used directly in the *SRC_URI*.

For a review of the skills used in this section see Sections 2.1.1 and 2.4.2.

5.1.5. Developing with 'devshell'

When debugging certain commands or even to just edit packages, the 'devshell' can be a useful tool. To start it you run a command like:

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

which will open a terminal with a shell prompt within the Poky environment. This means PATH is setup to include the cross toolchain, the pkgconfig variables are setup to find the right .pc files, configure will be able to find the Poky site files etc. Within this environment, you can run configure or compile command as if they were being run by Poky itself. You are also changed into the source (*S*) directory automatically. When finished with the shell just exit it or close the terminal window.

The default shell used by devshell is the gnome-terminal. Other forms of terminal can also be used 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 to make interaction with bitbakes multiple threads easier and also allow a client/server split of bitbake in the future (devshell will still work over X11 forwarding or similar).

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** and the same applies to other applications from gcc, bintuils, libtool etc. 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 which pulls from an external SCM it is possible to have Poky notice new changes added to the SCM and then build the latest version. This only works for SCMs where its possible to get a sensible revision number for changes. Currently it works for svn, git and bzr repositories.

To enable this behaviour it is simply a case of adding *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

GDB (The GNU Project Debugger) allows you to examine running programs to understand and fix problems and also to perform postmortem style analsys of program crashes. It is available as a package within poky and installed by default in sdk images. It works best when -dbg packages for the application being debugged are installed as the extra symbols give more meaningful output from GDB.

Sometimes, due to memory or disk space constraints, it is not possible to use GDB directly on the remote target to debug applications. This is due to the fact that GDB needs to load the debugging information and the binaries of the process being debugged. GDB then needs to perform many computations to locate information such as function names, variable names and values, stack traces, etc. even before starting the debugging process. This places load on the target system and can alter the characteristics of the program being debugged.

This is where GDBSERVER comes into play as it runs on the remote target and does not load any debugging information from the debugged process. Instead, the debugging information processing is done by a GDB instance running on a distant 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 some memory regions of that debugged program. All the debugging information loading and processing as well as the heavy debugging duty is done by the host GDB, giving the GDBSERVER running on the target a chance to remain small and fast.

As the host GDB is responsible for loading the debugging information and doing the necessary processing to make actual debugging happen, the user has to make sure it can access the unstripped binaries complete with their debugging information and compiled with no optimisations. The host GDB must also have local access to all the libraries used by the debugged program. On the remote target the binaries can remain stripped as GDBSERVER does not need any debugging information there. However they must also be compiled without optimisation matching the host's binaries.

The binary being debugged on the remote target machine is hence referred to as the 'inferior' in keeping with GDB documentation and terminology. Further documentation on GDB, is available on on their site.

5.2.1. Launching GDBSERVER on the target

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

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
```

After that, gdbserver should be listening on port 2345 for debugging commands coming from a remote GDB process running on the host computer. Communication between the GDBSERVER and the host GDB will be 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, described in the following sections.

5.2.2.1. Build the cross GDB package

A suitable gdb cross binary is required which runs on your host computer but knows about the the ABI of the remote target. This can be obtained from the the Poky toolchain, e.g. `/usr/local/poky/eabi-glibc/arm/bin/arm-poky-linux-gnueabi-gdb` which "arm" is the target architecture and "linux-gnueabi" the target ABI.

Alternatively this can be built directly by Poky. To do this you would build the gdb-cross package so for example you would run:

```
bitbake gdb-cross
```

Once built, the cross gdb binary can be found at

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

5.2.2.2. Making the inferior binaries available

The inferior binary needs to be available to GDB complete with all debugging symbols in order to get the best possible results along with any libraries the inferior depends on and their debugging symbols. There are a number of ways this can be done.

Perhaps the easiest is to have an 'sdk' image corresponding to the plain image installed on the device. In the case of 'pky-image-sato', 'poky-image-sdk' would contain suitable symbols. The sdk images already have the debugging symbols installed so its just a question expanding the archive to some location and telling GDB where this is.

Alternatively, poky can build a custom directory of files for a specific debugging purpose by reusing its tmp/rootfs directory, on the host computer in a slightly different way to normal. This directory contains the contents of the last built image. This process assumes the image running on the target was the last image to be built by Poky, the package *foo* contains the inferior binary to be debugged has been built without without optimisation and has debugging information available.

Firstly you want to install the *foo* package to tmp/rootfs by doing:

```
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
```

then,

```
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
```

which installs the debugging information too.

5.2.2.3. Launch the host GDB

To launch the host GDB, run the cross gdb binary identified above with the inferior binary specified on the commandline:

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

This loads the binary of program *foo* as well as its debugging information. Once the gdb prompt appears, you must instruct GDB to load all the libraries of the inferior from tmp/rootfs:

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

where `/path/to/tmp/rootfs` must be the absolute path to `tmp/rootfs` or wherever the binaries with debugging information are located.

Now, tell GDB to connect to the GDBSERVER running on the remote target:

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

Where remote-target-ip-address is the IP address of the remote target where the GDBSERVER is running. 2345 is the port on which the GDBSERVER is running.

5.2.2.4. Using the Debugger

Debugging can now proceed as normal, as if the debugging were being done on the local machine, for example to tell GDB to break in the *main* function, for instance:

```
break main
```

and then to tell GDB to "continue" the inferior execution,

```
continue
```

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

5.3. Profiling with OProfile

OProfile is a statistical profiler well suited to finding performance bottlenecks in both userspace software and the kernel. It provides answers to questions like "Which functions does my application spend the most time in when doing X?". Poky is well integrated with OProfile to make profiling applications on target hardware straightforward.

To use OProfile you need an image with 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 will take place. This can be achieved with "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 hence should be compiled with the "-fno-omit-framepointer" flag. In Poky this can be achieved with *SELECTED_OPTIMIZATION* = "-fexpensive-optimizations -fno-omit-framepointer -frename-registers -O2" or by setting *DEBUG_BUILD* = "1" in local.conf (the latter will also add extra debug information making the debug packages large).

5.3.1. Profiling on the target

All the profiling work can be performed on the target device. A simple OProfile session might look like:

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

Here, the reset command clears any previously profiled data, OProfile is then started. The options used to start OProfile mean dynamic library data is kept separately per application, kernel profiling is disabled and callgraphing is enabled up to 5 levels deep. 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). The profile is then stopped and the results viewed with opreport with options to see the separate library symbols and callgraph information.

Callgraphing means OProfile not only logs infomation about which functions time is being spent in but also which functions called those functions (their parents) and which functions that function calls (its children). The higher the callgraphing depth, the more accurate the results but this also increased the loging overhead so it should be used with caution. On ARM, binaries need to have the frame pointer enabled for callgraphing to work (compile with the gcc option -fno-omit-framepointer).

For more information on using OProfile please 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 the `opimport` program is needed. This is not included in standard Debian OProfile packages but an OProfile package with this addition is also available from the OpenedHand repository. We recommend using OProfile 0.9.3 or greater. Other patches to OProfile may be needed for recent OProfileUI features, but Poky usually includes all needed patches on the target device. Please see the OProfileUI README for up to date information, and the OProfileUI website for more information on the OProfileUI project.

5.3.2.1. Online mode

This assumes a working network connection with the target hardware. In this case you just need to run **"oprofile-server"** on the device. By default it listens on port 4224. This can be changed with the *--port* command line option.

The client program is called **oprofile-viewer**. The UI is relatively straightforward, the key functionality is accessed through the buttons on the toolbar (which are duplicated in the menus.) These buttons are:

- Connect - connect to the remote host, the IP address or hostname for the target can be supplied here.
- Disconnect - disconnect from the target.
- Start - start the profiling on the device.
- Stop - stop the profiling on the device and download the data to the local host. This will generate the profile and show it in the viewer.
- Download - download the data from the target, generate the profile and show it in the viewer.
- Reset - reset the sample data on the device. This will remove the sample information that was collected on a previous sampling run. Ensure you do this if you do not want to include old sample information.
- Save - save the data downloaded from the target to another directory for later examination.
- Open - load data that was previously saved.

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

Downloaded archives are kept in /tmp and cleared up when they are no longer in use.

If you wish to profile into the kernel, this is possible, you just need to ensure a vmlinux file matching the running kernel is available. In Poky this 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 ensuring a matching package is installed (**opkg install kernel-vmlinux**. These are automatically installed into development and profiling images alongside OProfile. There is a configuration option within the OProfileUI settings page where the location of the vmlinux file can be entered.

Waiting for debug symbols to transfer from the device can be slow and it's not always necessary to actually have them on device for OProfile use. All that is needed is a copy of the filesystem with the debug symbols present on the viewer system. The GDB remote debug section covers how to create such a directory with Poky and the location of this directory can again be specified in the OProfileUI settings dialog. If specified, it will be used where the file checksums match those on the system being profiled.

5.3.2.2. Offline mode

If no network access to the target is available an archive for processing in 'oprofile-viewer' can be generated with the following set of command.

```
# opcontrol --reset
# opcontrol --start --separate=lib --no-vmlinux -c 5
```

```
[do whatever is being profiled]
# opcontrol --stop
# oparchive -o my_archive
```

Where my_archive is the name of the archive directory where you would like the profile archive to be kept. The directory will be created for you. This can then be copied to another host and loaded using 'oprofile-viewer''s open functionality. The archive will be converted if necessary.

Appendix A. Reference: Directory Structure

Poky consists of several components and understanding what these are and where they're located is one of the keys to using it. This section walks through the Poky directory structure giving information about the various files and directories.

A.1. Top level core components

A.1.1. `bitbake/`

A copy of BitBake is included within Poky for ease of use, and should usually match the current BitBake stable release from the BitBake project. Bitbake, a metadata interpreter, reads the Poky metadata and runs the tasks defined in the Poky metadata. Failures are usually from the metadata, not BitBake itself, so most users don't 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 please 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, see the section seperate output directory.

A.1.3. `meta/`

This directory contains the core metadata, a key part of Poky. Within this directory there are definitions of the machines, the Poky distribution and the packages that make up a given system.

A.1.4. `meta-extras/`

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

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

These directories are optional layers to be added to core metadata, which are enabled by adding them to conf/bblayers.conf.

A.1.6. `scripts/`

This directory contains various integration scripts which implement extra functionality in the Poky environment, such as the QEMU scripts. This directory is appended to the PATH environment variable by the poky-init-build-env script.

A.1.7. `sources/`

While not part of a checkout, Poky will create this directory as part of any build. Any downloads are placed in this directory (as specified by the *DL_DIR* variable). This directory can be shared between Poky builds to save downloading files multiple times. SCM checkouts are also stored here as e.g. `sources/svn/` , `sources/cvs/` or `sources/git/` and the sources directory may contain archives of checkouts for various revisions or dates.

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

This location can be overridden by setting *DL_DIR* in `local.conf` . This directory can be shared between builds and even between machines via NFS, so downloads are only made once, speeding up builds.

A.1.8. `documentation`

This is the location for documentaiton about poky including this handbook.

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

This script is used to setup 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 use this before running Poky commands. Internally it uses scripts within the `scripts/` directory to do the bulk of the work. This script supports specifying any directory as the build output:

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

The above command can be typed from any directory, as long as POKY_SRC points to the desired Poky source tree. The optional BUILDDIR could be any directory you'd like Poky to generate the build output into.

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

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 hardcoded within Poky (e.g. by using '=' instead of '?='). Some variables are hardcoded 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 downloaded files should go (*DL_DIR*).

A.2.2. `build/conf/bblayers.conf`

This file defines layers walked by bitbake. If there's no `bblayers.conf` present, it is created from `bblayers.conf.sample` when the environment setup script is sourced.

A.2.3. `build/tmp/`

This is created by BitBake if it doesn't exist and is where all the Poky output is placed. To clean Poky and start a build from scratch (other than downloads), you can wipe this directory. 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 which can be used when subsequently running commands. These are stored here on a per machine basis.

A.2.5. `build/tmp/deploy/`

Any 'end result' output from Poky is placed under here.

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

Any .deb packages emitted by Poky are placed here, sorted into feeds for different architecture types.

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

Any .rpm packages emitted by Poky are placed here, sorted into feeds for different architecture types.

A.2.8. `build/tmp/deploy/images/`

Complete filesystem images are placed here. If you want to flash the resulting image from a build onto a device, look here for them.

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

Any resulting .ipk packages emitted by Poky are placed here.

A.2.10. `build/tmp/sysroots/`

Any package needing to share output with other packages does so within sysroots. This means it contains any shared header files and any shared libraries amongst other data. It is subdivided by architecture so multiple builds can run within the one build directory.

A.2.11. `build/tmp/stamps/`

This is used by BitBake for accounting purposes to keep track of which tasks have been run and when. It is also subdivided by architecture. The files are empty and the important information is the filenames and timestamps.

A.2.12. `build/tmp/log/`

This contains some general logs if not placing in a package's *WORKDIR*, such as the log output from check_pkg or distro_check tasks.

A.2.13. `build/tmp/pkgdata/`

This is an intermediate place for saving packaging data, which will be used in later packaging process. For detail please refer to package.bbclass.

A.2.14. `build/tmp/pstagelogs/`

This directory contains manifest for task based prebuilt. Each manifest is basically a file list for installed files from a given task, which would be useful for later packaging or cleanup process.

A.2.15. `build/tmp/work/`

This directory contains various subdirectories for each architecture, and each package built by BitBake has its own work directory under the appropriate architecture subdirectory. All tasks are executed from this work directory. As an example, the source for a particular package will be unpacked, patched, configured and compiled all within its own work directory.

It is worth considering the structure of a typical work directory. An example is 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 is *WORKDIR*`/temp/` which has log files for each task (`log.do_*.pid`) and the scripts BitBake runs for each task (`run.do_*.pid`). The *WORKDIR*`/image/` directory is where **make install** places its output which is then split into subpackages within *WORKDIR* `/packages-split/`.

A.3. `meta/` - The Metadata

As mentioned previously, this is the core of Poky. It has several important subdivisions:

A.3.1. `meta/classes/`

Contains the `*.bbclass` files. Class files are used to abstract common code allowing it to be reused by multiple packages. The `base.bbclass` file is inherited by every package. Examples of other important classes are `autotools.bbclass` that in theory allows any Autotool-enabled package to work with Poky with minimal effort, or `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 (`image.bbclass` , `rootfs_*.bbclass` and `package*.bbclass`).

A.3.2. `meta/conf/`

This is the core set of configuration files which start from `bitbake.conf` and from which all other configuration files are included (see the includes at the end of the file, even `local.conf` is loaded from there!). While `bitbake.conf` sets up the defaults, these can often be overridden by user (`local.conf`), machine or distribution configuration files.

A.3.3. `meta/conf/machine/`

Contains all the machine configuration files. If you set MACHINE="spitz", the end result is Poky looking 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, this is the directory to look in.

A.3.4. `meta/conf/distro/`

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

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

Anything linking to specific hardware or hardware configuration information are placed here, such as uboot, grub, etc.

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

Libraries and applications related to communication with other devices

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

What's needed to build a basic working Linux image including commonly used dependencies

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

Tools primarily used by the build system (but can also be used on targets)

A.3.9. `meta/recipes-extended/`

Applications which whilst not essential add features compared to the alternatives in core. May be needed for full tool functionality or LSB compliance.

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

All things related to the GTK+ application framework

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

X and other graphically related system libraries

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

The kernel and generic applications/libraries with strong kernel dependencies

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

Codecs and support utilties for audio, images and video

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

All things related to the QT application framework

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

The Sato demo/reference UI/UX, its associated apps and configuration

A.3.16. `meta/site/`

Certain autoconf test results cannot be determined when cross compiling since it can't run tests on a live system. This directory therefore contains a list of cached results for various architectures which is passed to autoconf.

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 type **bitbake poky-image-sato**. This section aims to give an overview of what happens behind the scenes from a BitBake perspective.

It is worth noting that bitbake aims to be a generic "task" executor capable of handling complex dependency relationships. As such it has no real knowledge of what the tasks it is executing actually do. It just considers a list of tasks with dependencies and handles metadata consisting of variables in a certain format which get passed to the tasks.

B.1. Parsing

The first thing BitBake does is work out its configuration by looking for a file called `bitbake.conf`. Bitbake searches through the `BBPATH` environment variable looking for a `conf/` directory containing a `bitbake.conf` file and adds the first `bitbake.conf` file found in `BBPATH` (similar to the PATH environment variable). For Poky, `bitbake.conf` is found in `meta/conf/`.

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 users 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 which 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. In particular, `base.bbclass` is always included, as will any other classes specified in the configuration using the *INHERIT* variable. Class files are searched for in a classes subdirectory under the paths in `BBPATH` in the same way as configuration files.

After the parsing of the configuration files is complete, the variable *BBFILES* is set, usually in

`local.conf`, and defines the list of places to search for `.bb` files. By default this 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 acheived 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 + 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.

Parsing `.bb` files is a time consuming process, so a cache is kept to speed up subsequent parsing. This cache is invalid if the timestamp of the `.bb` file itself has changed, or if the timestamps of any of the include, configuration or class files the `.bb` file depends on have changed.

B.2. Preferences and Providers

Once all the `.bb` files have been parsed, BitBake will proceed to build "poky-image-sato" (or whatever was specified on the commandline) 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 would lead to packages like Contacts, Dates, BusyBox and these in turn depend on glibc and the toolchain.

Sometimes a target might have multiple providers and a common example is "virtual/kernel" that is provided by each kernel package. Each machine will often elect the best provider of its kernel with a line like 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 complicated by the fact multiple versions might be present. Bitbake defaults to the highest version of a provider by default. Version comparisons are made using the same method as Debian. The *PREFERRED_VERSION* variable can be used to specify a particular version (usually in the distro configuration) but the order can also be influenced by the *DEFAULT_PREFERENCE* variable. By default files have a preference of "0". Setting the *DEFAULT_PREFERENCE* to "-1" will make a package unlikely to be used unless it was explicitly referenced and "1" makes it likely the package will be used. *PREFERRED_VERSION* overrides any *DEFAULT_PREFERENCE*. *DEFAULT_PREFERENCE* is often used to mark more experimental new versions of packages until they've undergone sufficient testing to be considered stable.

The end result is that internally, BitBake has now built a list of providers for each target it needs in order of priority.

B.3. Dependencies

Each target BitBake builds consists of multiple tasks (e.g. fetch, unpack, patch, configure, compile etc.). For best performance on multi-core systems, BitBake considers each task as an independent entity with a set of dependencies. There are many variables that are used to signify these dependencies and more information can be found about these in the BitBake manual. At a basic level it is sufficient to know that BitBake uses the *DEPENDS* and *RDEPENDS* variables when calculating dependencies and descriptions of these variables are available through the links.

B.4. The Task List

Based on the generated list of providers and the dependency information, BitBake can now calculate exactly which tasks it needs to run and in what order. The build now starts with BitBake forking off threads up to the limit set in the *BB_NUMBER_THREADS* variable as long as there are tasks ready to run, i.e. tasks with all their dependencies met.

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 will not rerun tasks its already completed unless a timestamp is found to be invalid. Currently, invalid timestamps are only considered on a per `.bb` file basis so if for example the configure stamp has a timestamp greater than the compile timestamp for a given target the compile task would rerun but this has no effect on other providers depending on that target. This could change or become configurable in future versions of BitBake. Some tasks are marked as "nostamp" tasks which means no timestamp file will be written and the task will always rerun.

Once all the tasks have been completed BitBake exits.

B.5. Running a Task

It's worth noting what BitBake does to run a task. A task can either be a shell task or a python task. For shell tasks, BitBake writes a shell script to `${WORKDIR}/temp/run.do_taskname.pid` and then executes the script. The generated shell script contains all the exported variables, and the shell functions with all variables expanded. Output from the shell script is sent to the file `${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.

Python functions are executed internally to BitBake itself and logging goes to the controlling terminal. Future versions of BitBake will write the functions to files in a similar way to shell functions and logging will also go to the log files in a similar way.

B.6. Commandline

To quote from "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
  -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
  -S, --dump-signatures
                      don't execute, just dump out the signature
                      construction information
  -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 EXTRA_ASSUME_PROVIDED, --ignore-deps=EXTRA_ASSUME_PROVIDED
                      Assume these dependencies don't exist and are already
                      provided (equivalent to ASSUME_PROVIDED). Useful to
                      make dependency graphs more appealing
  -l DEBUG_DOMAINS, --log-domains=DEBUG_DOMAINS
                      Show debug logging for the specified logging domains
  -P, --profile       profile the command and print a report
  -u UI, --ui=UI      userinterface to use
  --revisions-changed Set the exit code depending on whether upstream
                      floating revisions have changed or not
```

B.7. Fetchers

As well as the containing the parsing and task/dependency handling code, bitbake also contains a set of "fetcher" modules which allow fetching of source code from various types of sources. Example sources might be from disk with the metadata, from websites, from remote shell accounts or from SCM systems like cvs/subversion/git.

The fetchers are usually triggered by entries in *SRC_URI*. Information about the options and formats of entries for specific fetchers can be found in the BitBake manual.

One useful feature for certain SCM fetchers is the ability to "auto-update" when the upstream SCM changes version. Since this requires certain functionality from the SCM only certain systems support it, currently Subversion, Bazaar and to a limited extent, Git. It works using the *SRCREV* variable. See the developing with an external SCM based project 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 usualy 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' regenearates 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. A full list of image targets can be found by looking in the directories `meta/recipes-core/images/`, `meta/recipes-extended/images/`, `meta/recipes-sato/images/` and `meta/recipes-tbd/meta/`. The standard images are listed below along with details of what they contain:

- *poky-image-minimal* - A small image, just enough to allow a device to boot

- *poky-image-base* - console only image with full support of target device hardware
- *poky-image-core* - X11 image with simple apps like terminal, editor and file manager
- *poky-image-sato* - X11 image with Sato theme and Pimlico applications. Also contains terminal, editor and file manager.
- *poky-image-sdk* - X11 image like poky-image-sato but also include native toolchain and libraries needed to build applications on the device itself. Also includes testing and profiling tools and debug symbols.
- *meta-toolchain* - This generates a tarball containing a standalone toolchain which can be used externally to Poky. It is self contained and unpacks to the `/opt/poky` directory. It also contains a copy of QEMU and the scripts neccessary to run poky QEMU images.
- *meta-toolchain-sdk* - This includes everything in meta-toolchain but also includes development headers and libraries forming a complete standalone SDK. See the Developing using the Poky SDK and Developing using the Anjuta Plugin sections 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

Table of Contents

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

Glossary

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

BBFILES

List of recipes used by BitBake to build software

BBINCLUDELOGS

Variable which controls how BitBake displays logs on build failure.

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 evalutates to match the machines the recipe works with. It stops recipes being run on machines they're incompatible with, which is partciuarly 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

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_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)

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.

PV

Version of package. The default value is "1.0"

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 sytems 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.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?

Poky is a derivative of OpenEmbedded, 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. 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.3. 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.4. Are there any products running poky ?

The Vernier 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.5. 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.6. 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.7. 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.8. What is GNOME Mobile? What's the difference between GNOME Mobile and GNOME?

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

H.9. 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.10. 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
- help2man from Karan repository

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.11. I see lots of 404 responses for files on http://pokylinux.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.12. 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.13. 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.14. 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.15. 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.16. I'm seeing random build failures. Help?!

If the same build is failing in totally different and random ways the most likely explaination 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 senstive to even single bit failure in any of these areas. Totally random failures have always been traced back to hardware or virtualisation issues.

H.17. What do we need to ship for licence complience?

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 complience 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.

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 in the bug tracker.

I.3. Mailing list

To subscribe to the mailing list send mail to:

```
poky+subscribe <at> openedhand <dot> com
```

Then follow the simple instructions in subsequent reply. Archives are available here.

I.4. IRC

Join #poky on freenode.

I.5. Links

- The Poky website
- OpenedHand - The original company behind Poky.
- Intel Corporation - The company who acquired OpenedHand in 2008.

- OpenEmbedded - The upstream generic embedded distribution Poky derives from (and contributes to).
- Bitbake - The tool used to process Poky metadata.
- Bitbake User Manual
- Pimlico - A suite of lightweight Personal Information Management (PIM) applications designed primarily for handheld and mobile devices.
- 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.pokylinux.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.

Index