Poky Handbook

Hitchhiker's Guide to Poky

Richard Purdie, OpenedHand Ltd **<richard@openedhand.com>**
Tomas Frydrych, OpenedHand Ltd **<tf@openedhand.com>**
Marcin Juszkiewicz, OpenedHand Ltd **<hrw@openedhand.com>**
Dodji Seketeli, OpenedHand Ltd **<dodji@openedhand.com>**

Poky Handbook: Hitchhiker's Guide to Poky

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

Table of Contents

Chapter 1. Introduction

1. What is Poky?

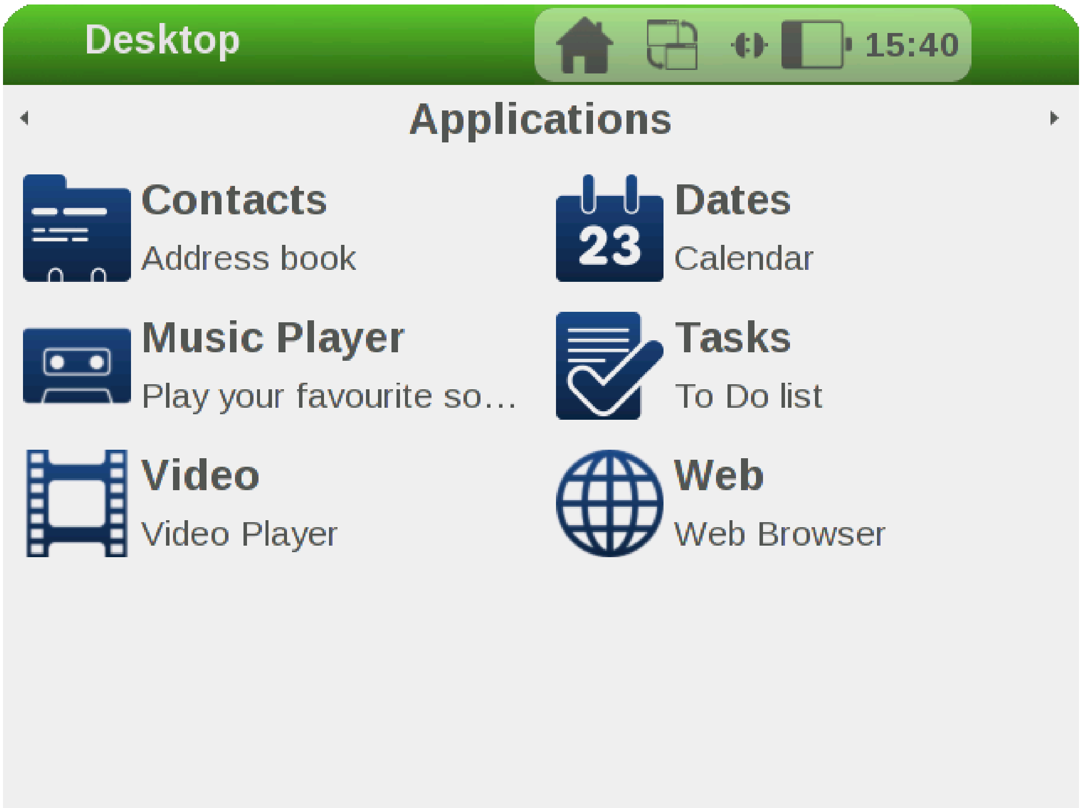
Poky is an open source platform build tool. It is a complete software development environment for the creation of Linux devices. It aids the design, development, building, debugging, simulation and testing of complete modern software stacks using Linux, the X Window System and GNOME Mobile based application frameworks. It is based on OpenEmbedded [http://openembedded.org/] but has been customised with a particular focus.

Poky was setup to:

- Provide an open source Linux, X11, Matchbox, GTK+, Pimlico, Clutter, and other GNOME Mobile [http://gnome.org/mobile] technologies based full platform build and development tool.
- Create a focused, stable, subset of OpenEmbedded that can be easily and reliably built and developed upon.
- Fully support a wide range of x86 and ARM 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 (both x86 and ARM) and the ARM based Sharp Zaurus series of devices. 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 backed up by commercial support provided by the principle developer and maintainer of Poky, OpenedHand Ltd [http://o-hand.com/].

2. Documentation Overview

The handbook 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 and debugging the Poky build system. 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 3.1 (Pinky).

3. System Requirements

We recommend Debian-based distributions, in particular a recent Ubuntu release (7.04 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.

On a Debian-based system, you need the following packages installed:

- build-essential
- python
- diffstat
- texinfo
- texi2html
- cvs
- subversion
- wget
- gawk
- help2man
- bochsbios (only to run qemux86 images)

Debian users can add debian.o-hand.com to their APT sources (See http://debian.o-hand.com for instructions on doing this) and then run "apt-get install qemu poky-depends poky-scripts" which will automatically install all these dependencies. OpenedHand can also provide VMware images with Poky and all dependencies pre-installed if required.

Poky can use a system provided QEMU or build its own depending on how it's configured. See the options in `local.conf` for more details.

4. Quick Start

4.1. Building and Running an Image

If you want to try Poky, you can do so in a few commands. The example below checks out the Poky source code, sets up a build environment, builds an image and then runs that image under the QEMU emulator in ARM system emulation mode:

```
$ wget http://pokylinux.org/releases/pinky-3.1.tar.gz
$ tar zxvf pinky-3.1.tar.gz
$ cd pinky-3.1/
$ source poky-init-build-env
$ bitbake poky-image-sato
$ runqemu qemuarm
```

Note

This process will need Internet access, about 3 GB of disk space available, and you should expect the build to take about 4 - 5 hours since it is building an entire Linux system from source including the toolchain!

To build for other machines see the MACHINE variable in build/conf/local.conf which also contains other configuration information. The images/kernels built by Poky are placed in the `tmp/deploy/images` directory.

You could also run "poky-qemu zImage-qemuarm.bin poky-image-sato-qemuarm.ext2" within the images directory if you have the poky-scripts Debian package installed from debian.o-hand.com. This allows the QEMU images to be used standalone outside the Poky build environment.

To setup networking within QEMU see the QEMU/USB networking with IP masquerading section.

4.2. Downloading and Using Prebuilt Images

Prebuilt images from Poky are also available if you just want to run the system under QEMU. To use these you need to:

- Add debian.o-hand.com to your APT sources (See http://debian.o-hand.com for instructions on doing this)
- Install patched QEMU and poky-scripts:

```
$ apt-get install qemu poky-scripts
```

- Download a Poky QEMU release kernel (*zImage*qemu*.bin) and compressed filesystem image (poky-image-*-qemu*.ext2.bz2) which you'll need to decompress with 'bzip2 -d'. These are available from the last release [http://pokylinux.org/releases/blinky-3.0/] or from the autobuilder [http://pokylinux.org/autobuild/poky/].
- Start the image:

```
$ poky-qemu <kernel> <image>
```

Note

A patched version of QEMU is required at present. A suitable version is available from http://debian.o-hand.com, it can be built by poky (bitbake qemu-native) or can be downloaded/built as part of the toolchain/SDK tarballs.

5. Obtaining Poky

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.

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://pokylinux.org/autobuild/ where the numbers represent the svn revision the builds were made from.

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 `/usr/local/poky`) and then enabling the option in `local.conf`.

5.3. Development Checkouts

Poky is available from our SVN repository located at http://svn.o-hand.com/repos/poky/trunk; a web interface to the repository can be accessed at http://svn.o-hand.com/view/poky/.

'trunk' 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.

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 'collection'.

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

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.

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 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.

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.

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. Running a Build

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

```
$ source poky-init-build-env
```

Once the Poky build environment is setup, 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.

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.

3.1. USB Networking

Devices commonly have USB connectivity. To connect to the usbnet interface, on the host machine run:

```
modprobe usbnet
ifconfig usb0 192.168.0.200
route add 192.168.0.202 usb0
```

3.2. QEMU/USB networking with IP masquerading

On Ubuntu, Debian or similar distributions you can have the network automatically configured. You can also enable masquerading between the QEMU system and the rest of your network. To do this you need to edit `/etc/network/interfaces` to include:

```
allow-hotplug tap0
iface tap0 inet static
        address 192.168.7.200
```

```
        netmask 255.255.255.0
        network 192.168.7.0
        post-up iptables -A POSTROUTING -t nat -j MASQUERADE -s 192.168.7.0/24
        post-up echo 1 > /proc/sys/net/ipv4/ip_forward
        post-up iptables -P FORWARD ACCEPT
```

This ensures the tap0 interface will be up everytime you run QEMU and it will have network/internet access.

Under emulation there are two steps to configure for internet access via tap0. The first step is to configure routing:

```
route add default gw 192.168.7.200 tap0
```

The second is to configure name resolution which is configured in the `/etc/resolv.conf` file. The simplest solution is to copy it's content from the host machine.

USB connections to devices can be setup and automated in a similar way. First add the following to `/etc/network/interfaces`:

```
allow-hotplug usb0
iface usb0 inet static
        address 192.168.0.200
        netmask 255.255.255.0
        network 192.168.0.0
        post-up iptables -A POSTROUTING -t nat -j MASQUERADE -s 192.168.0.0/24
        post-up echo 1 > /proc/sys/net/ipv4/ip_forward
        post-up iptables -P FORWARD ACCEPT
```

and then to configure routing on the device you would use:

```
route add default gw 192.168.0.202 usb0
```

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. Its impossible to detail every possible potential failure here but there are some general tips to aid debugging:

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.

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 the 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.

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.

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.

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.

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.

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.

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.

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

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

To build an application from a single file stored locally 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"

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" and "helloworld-dbg" packages will be built.

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).

```
DESCRIPTION = "GNU Helloworld application"
SECTION = "examples"
LICENSE = "GPLv2"

SRC_URI = "${GNU_MIRROR}/hello/hello-${PV}.tar.bz2"

inherit autotools
```

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.

```
DESCRIPTION = "Tools for managing memory technology devices."
SECTION = "base"
DEPENDS = "zlib"
HOMEPAGE = "http://www.linux-mtd.infradead.org/"
LICENSE = "GPLv2"

SRC_URI = "ftp://ftp.infradead.org/pub/mtd-utils/mtd-utils-${PV}.tar.gz"

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

do_install() {
        oe_runmake install DESTDIR=${D}
}
```

1.4. Controlling packages content

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

Below the "libXpm" recipe 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"
```

```
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.

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. 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 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.

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.

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 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"
```

2.2. Customising Images through custom tasks

For for complex custom images, the best approach is to create a custom task package which is them 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 complimentary -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 tasks 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.

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 based on this generates the IMAGE_INSTALL variable automatically. Extra features can be added by extending the class or creating a custom class for use with specialised image .bb files.

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 and the the image built.

```
bitbake -cclean 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. 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. 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.

3.1. 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.

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 packages/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_MACHINES .

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`.

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 "collections" 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.

4.1. Bitbake Collections

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 collections which provide a way to handle this which is fully supported and actively encouraged within Poky.

In the standard tree, meta-extras is an example of how you can do this. As standard the data in meta-extras is not used on a Poky build but local.conf.sample shows how to enable it:

```
BBFILES := "${OEROOT}/meta/packages/*/*.bb ${OEROOT}/meta-extras/packages/*/*.bb"
BBFILE_COLLECTIONS = "normal extras"
BBFILE_PATTERN_normal = "^${OEROOT}/meta/"
BBFILE_PATTERN_extras = "^${OEROOT}/meta-extras/"
BBFILE_PRIORITY_normal = "5"
BBFILE_PRIORITY_extras = "5"
```

As can be seen, the extra recipes are added to BBFILES. The BBFILE_COLLECTIONS variable is then set to contain a list of collection names. The BBFILE_PATTERN variables are regular expressions used to match files from BBFILES into a particular collection in this case by using the base pathname. The BBFILE_PRIORITY variable then assigns the different priorities to the files in different collections. This is useful in situations where the same package might appear in both repositories and allows you to choose which collection should 'win'.

This works well for recipes. For bbclasses and configuration files, you can use 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.

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 that OpenedHand uses for commits to Poky.

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.

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.

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 not be added into new recipes as this is default value. When upgrading the version of a package (PV), the PR variable should be removed.

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 [http://www.debian.org/doc/debian-policy/ch-controlfields.html] 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 ipkg update;ipkg upgrade commands (or similar for dpkg/apt or rpm based systems). The aim is to ensure Poky has upgradable packages in all cases.

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 --cmd compile --force NAME_OF_PACKAGE
```

Other tasks may also be called this way.

5.1. Modifying Package Source Code with quilt

By default Poky uses quilt [http://savannah.nongnu.org/projects/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;patch=1"
```

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

Chapter 4. Platform Development with Poky

1. Software development

Poky supports several methods of software development. These different forms of development are explained below and can be switched between as needed.

1.1. Developing externally using the Poky SDK

The meta-toolchain and meta-toolchain-sdk targets (see the images section) build tarballs which contain toolchains and libraries suitable for application development outside Poky. These unpack into the `/usr/local/poky` directory and contain a setup script, e.g. `/usr/local/poky/eabi-glibc/arm/environment-setup` which can be sourced to initialise a suitable environment. After sourcing this, the compiler, QEMU scripts, QEMU binary, a special version of pkgconfig and other useful utilities are added to the PATH. Variables to assist pkgconfig and autotools are also set so that, for example, configure can find pre-generated test results for tests which need target hardware to run.

Using the toolchain with autotool enabled packages is straightforward, just pass the appropriate host option to configure e.g. "./configure --host=arm-poky-linux-gnueabi". For other projects it is usually a case of ensuring the cross tools are used e.g. CC=arm-poky-linux-gnueabi-gcc and LD=arm-poky-linux-gnueabi-ld.

1.2. Developing externally using the Anjuta plugin

An Anjuta IDE plugin exists to make developing software within the Poky framework easier for the application developer. It presents a graphical IDE from which the developer 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 along with Anjuta and the Anjuta plugin. The Poky Anjuta plugin is available from the OpenedHand SVN repository located at http://svn.o-hand.com/repos/anjuta-poky/trunk/anjuta-plugin-sdk/; a web interface to the repository can be accessed at http://svn.o-hand.com/view/anjuta-poky/. See the README file contained in the project for more information about the dependencies and how to get them along with details of the prebuilt packages.

1.2.1. Setting up the Anjuta plugin

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

To use the plugin, first open or create an existing project. If 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.

To activate the plugin go to Edit → Preferences, then choose General from the left hand side. Choose the Installed plugins tab, scroll down to Poky SDK and check the box. The plugin is now activated but first it must be configured.

1.2.2. Configuring the Anjuta plugin

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

- SDK root: this is the root directory of the SDK for an ARM EABI SDK this will be `/usr/local/poky/eabi-glibc/arm`. This directory will contain directories named like "bin", "include", "var", etc. With the file chooser it is important to enter into the "arm" subdirectory for this example.
- Toolchain triplet: this is the cross compile triplet, e.g. "arm-poky-linux-gnueabi".
- Kernel: use the file chooser to select the kernel to use with QEMU
- Root filesystem: use the file chooser to select the root filesystem image, this should be an image (not a tarball)

1.2.3. Using the Anjuta plugin

As an example, cross-compiling a project, deploying it into QEMU and running a debugger against it and then doing a system wide profile.

Choose Build → Run Configure or Build → Run Autogenerate to run "configure" (or to run "autogen") for the project. This passes command line arguments to instruct it to cross-compile.

Next do 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 may find that your project fails to link. Simply do Build → Clean Project to remove the old binaries. You may then try building again.

Next start QEMU by using Tools → Start QEMU, this will start QEMU and will show any error messages in the message view. Once Poky has fully booted within QEMU you may now deploy into it.

Once built and QEMU is running, choose Tools → Deploy, this will install the package into a temporary directory and then copy using rsync over SSH into the target. Progress and messages will be shown in the message view.

To debug a program installed into onto the target choose Tools → Debug remote. This prompts 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 may stop the debugger on the target using Tools → Stop debugger.

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 will open a terminal with the SSH command inside.

To do a system wide profile against the system running in QEMU choose Tools → Profile remote. This will start up OProfileUI with the appropriate parameters to connect to the server running inside QEMU and will also supply the path to the debug information necessary to get a useful profile.

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 `/usr/local/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.

1.4. Developing externally in a chroot

If you have a system that matches the architecture of the Poky machine you're using, such as qemux86, you can run binaries directly from the image on the host system using a chroot combined with tools like Xephyr [http://projects.o-hand.com/xephyr].

Poky has some scripts to make using its qemux86 images within a chroot easier. To use these you need to install the poky-scripts package or otherwise obtain the `poky-chroot-setup` and `poky-chroot-run` scripts. You also need Xephyr and chrootuid binaries available. To initialize a system use the setup script:

```
# poky-chroot-setup <qemux86-rootfs.tgz> <target-directory>
```

which will unpack the specified qemux86 rootfs tarball into the target-directory. You can then start the system with:

```
# poky-chroot-run <target-directory> <command>
```

where the target-directory is the place the rootfs was unpacked to and command is an optional command to run. If no command is specified, the system will drop you within a bash shell. A Xephyr window will be displayed containing the emulated system and you may be asked for a password since some of the commands used for bind mounting directories need to be run using sudo.

There are limits as to how far the the realism of the chroot environment extends. It is useful for simple development work or quick tests but full system emulation with QEMU offers a much more realistic environment for more complex development tasks. Note that chroot support within Poky is still experimental.

1.5. 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.

1.6. 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.

1.7. 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.

2. Debugging with GDB Remotely

GDB [http://sourceware.org/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 [http://sourceware.org/gdb/documentation/].

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.

2.2. Launching GDB on the host computer

Running GDB on the host computer takes a number of stages, described in the following sections.

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/cross/bin/<target-abi>-gdb
```

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/staging/i686-linux/usr/bin/ipkg-cl -f \
tmp/work/<target-abi>/poky-image-sato-1.0-r0/temp/ipkg.conf -o \
tmp/rootfs/ update
```

then,

```
tmp/staging/i686-linux/usr/bin/ipkg-cl -f \
tmp/work/<target-abi>/poky-image-sato-1.0-r0/temp/ipkg.conf \
-o tmp/rootfs install foo

tmp/staging/i686-linux/usr/bin/ipkg-cl -f \
tmp/work/<target-abi>/poky-image-sato-1.0-r0/temp/ipkg.conf \
```

```
-o tmp/rootfs install foo-dbg
```

which installs the debugging information too.

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.

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/.

3. Profiling with OProfile

OProfile [http://oprofile.sourceforge.net/] 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).

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/.

3.2. Using OProfileUI

A graphical user interface for OProfile is also available. You can either use prebuilt Debian packages from the OpenedHand repository [http://debian.o-hand.com/] or download and build 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 [http://debian.o-hand.com/]. 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 [http://svn.o-hand.com/repos/oprofileui/trunk/README] for up to date information, and the OProfileUI website [http://labs.o-hand.com/oprofileui] for more information on the OProfileUI project.

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 (ipkg 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.

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.

1. Top level core components

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/.

1.2. `build/`

This directory contains user configuration files and the output from Poky.

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.

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.

1.5. `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.

1.6. `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.

1.7. `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.

2. `build/` - The Build Directory

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).

2.2. `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.

2.3. `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.

2.4. `build/tmp/cross/`

The cross compiler when generated is placed into this directory and those beneath it.

2.5. `build/tmp/deploy/`

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

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

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

2.7. `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.

2.8. `build/tmp/deploy/ipk/`

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

2.9. `build/tmp/rootfs/`

This is a temporary scratch area used when creating filesystem images. It is run under fakeroot and is not useful once that fakeroot session has ended as information is lost. It is left around since it is still useful in debugging image creation problems.

2.10. `build/tmp/staging/`

Any package needing to share output with other packages does so within staging. 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.

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.

2.12. `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`/install/`.

3. `meta/` - The Metadata

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

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`).

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.

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.

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.

3.5. meta/packages/

Each application (package) Poky can build has an associated .bb file which are all stored under this directory. Poky finds them through the BBFILES variable which defaults to packages/*/*.bb. Adding a new piece of software to Poky consists of adding the appropriate .bb file. The .bb files from OpenEmbedded upstream are usually compatible although they are not supported.

3.6. 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 a program written in Python which 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 its 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.

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. If multiple directories are specified a system referred to as "collections" is used to determine which files have priority.

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.

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-oh.bb` and `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.

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 found about these in the BitBake manual [http://bitbake.berlios.de/manual/]. At a basic level it is sufficient to know that BitBake uses the DEPENDS and RDEPENDS variables when calculating dependencies and descriptions of these variables are available through the links.

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 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.

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.

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.
  -f, --force           force run of specified cmd, regardless of stamp status
  -i, --interactive     drop into the interactive mode also called the BitBake
                        shell.
  -c CMD, --cmd=CMD     Specify task to execute. Note that this only executes
                        the specified task for the providee and the packages
                        it depends on, i.e. 'compile' does not implicitly call
                        stage for the dependencies (IOW: use only if you know
                        what you are doing). Depending on the base.bbclass a
                        listtasks tasks is defined and will show available
                        tasks
  -r FILE, --read=FILE  read the specified file before bitbake.conf
  -v, --verbose         output more chit-chat to the terminal
  -D, --debug           Increase the debug level. You can specify this more
                        than once.
  -n, --dry-run         don't execute, just go through the motions
  -p, --parse-only      quit after parsing the BB files (developers only)
  -d, --disable-psyco   disable using the psyco just-in-time compiler (not
```

```
                        recommended)
-s, --show-versions     show current and preferred versions of all packages
-e, --environment       show the global or per-package environment (this is
                        what used to be bbread)
-g, --graphviz          emit the dependency trees of the specified packages in
                        the dot syntax
-I IGNORED_DOT_DEPS, --ignore-deps=IGNORED_DOT_DEPS
                        Stop processing at the given list of dependencies when
                        generating dependency graphs. This can help to make
                        the graph more appealing
-l DEBUG_DOMAINS, --log-domains=DEBUG_DOMAINS
                        Show debug logging for the specified logging domains
-P, --profile           profile the command and print a report
```

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

One useful feature for certain SCM fetchers is the ability to "auto-update" when the upstream SCM changes version. Since this 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 `build/` directory 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.

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 contains some commonly some commonly used functions such as `oe_libinstall` and `oe_runmake`. The end of the class file has a list of standard mirrors for software projects for use by the fetcher code.

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.

Its useful to have some idea of the tasks this class defines work and what they do behind the scenes.

- 'do_configure' regenearates the configure script 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.

By default the class does not stage headers and libraries so the recipe author needs to add their own `do_stage()` task. For typical recipes the following example code will usually be enough:

```
do_stage() {
autotools_stage_all
}
```

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

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.

5. Binary config scripts - **binconfig.bbclass**

Before pkg-config became 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 `staging/` directory. It also changes all paths to point into the `staging/` directory so all builds which use the script will use the correct directories for the cross compiling layout.

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'.

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 `staging/` directory. By making use of sysroot functionality within pkgconfig this class no longer has to manipulate the files.

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

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.

10. Python extensions - `distutils.bbclass`

Recipes for Python extensions 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.

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

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

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.

12. Packaging - `package*.bbclass`

The packaging classes add support for generating packages from the output from builds. 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 and package_ipk.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.

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

This class handle 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.

The means that each kerel module built is packaged separately and inter-modules dependencies are created by parsing the modinfo output. If all modules are required then installing "kernel-modules" package will install all packages with modules and various other kernel packages such as "kernel-vmlinux" are also generated.

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.

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.

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

This class checks prerequisite software is present to try and identify and notify the user of problems which will affect their build. It also performs basic checks of the users configuration from local.conf to prevent common mistakes and resulting build failures. Its usually up to distribution policy to include this class (Poky does).

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 this makes, checking for common problems which break builds/packages/images, see the bbclass file for more information. Its usually up to distribution policy to include this class (Poky doesn't at the time of writing but plans to soon).

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.

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 `meta/packages/images/` directory. 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 `/usr/local/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.

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)

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

3. Reference: Images

The contents of images generated by Poky can be controlled by the IMAGE_FEATURES variable in local.conf. Through this you can add several different predefined packages such as development utilities or packages with debug information needed to investigate application problems or profile applications.

Current list of IMAGE_FEATURES contains:

- apps-console-core - Core console applications such as ssh daemon, avahi daemon, portmap (for mounting NFS shares)
- x11-base - X11 server + minimal desktop
- x11-sato - OpenedHand Sato environment
- apps-x11-core - Core X11 applications such as an X Terminal, file manager, file editor
- apps-x11-games - A set of X11 games
- apps-x11-pimlico - OpenedHand Pimlico application suite
- tools-sdk - A full SDK which runs on device
- tools-debug - Debugging tools such as strace and gdb
- tools-profile - Profiling tools such as oprofile, exmap and LTTng

- tools-testapps - Device testing tools (e.g. touchscreen debugging)
- nfs-server - NFS server (exports / over NFS to everybody)
- dev-pkgs - Development packages (headers and extra library links) for all packages installed in a given image
- dbg-pkgs - Debug packages for all packages installed in a given image

Appendix F. Reference: Variables Glossary

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

Glossary

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

A

AUTHOR	E-mail address to contact original author(s) - to send patches, forward bugs...
AUTOREV	Use current (newest) source revision - used with SRCREV variable.

B

BB_NUMBER_THREADS	Number of BitBake threads
BBFILES	List of recipes used by BitBake to build software
BBINCLUDELOGS	Variable which controls how BitBake displays logs on build failure.

C

CFLAGS	Flags passed to C compiler for the target system. Evaluates to the same as TARGET_CFLAGS.
COMPATIBLE_MACHINES	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 to increase parsing speed as if its found the current machine is not compatible, further parsing of the recipe is skipped.
CONFIG_SITE	Contains a list of files which containing autoconf test results relevant to the current build. This variable is used by the autotools utilities when running configure.
CVS_TARBALL_STASH	Location to search for pre-generated tarballs when fetching from remote SCM repositories (CVS/SVN/GIT)

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_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_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

LICENSE	List of package source licenses.

M

MACHINE	Target device
MACHINE_ESSENTIAL_RDEPENDS	List of packages required to boot device
MACHINE_ESSENTIAL_RRECOOMENDS	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 ${PN}-dev"
PN	Name of package.
PR	Revision of package.
PV	Version of package. The default value is "1.0"
PE	Epoch of the package. The default value is "1". The field is used to make upgrades possible when the versioning scheme changes in some backwards incompatible way.
PREFERRED_PROVIDER	If multiple recipes provide an item, this variable determines which one should be given preference. It should be set to the "$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.
POKYLIBC	Libc implementation selector - glibc 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 which conflict with this one. Package will not be installed if they will not be 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 namespave 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 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. It defaults to SHELLRCCMD.
SHELLRCCMD	How to launch a shell, defaults to bash.
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.

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.

1. Distro Configuration

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

2. Machine Configuration

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

3. Local Configuration (local.conf)

- DISTRO
- MACHINE
- DL_DIR
- BBFILES
- IMAGE_FEATURES
- PACKAGE_CLASSES
- BB_NUMBER_THREADS
- BBINCLUDELOGS
- CVS_TARBALL_STASH
- ENABLE_BINARY_LOCALE_GENERATION

4. Recipe Variables - Required

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

5. Recipe Variables - Dependencies

- DEPENDS
- RDEPENDS
- RRECOMMENDS
- RCONFLICTS
- RREPLACES

6. Recipe Variables - Paths

- WORKDIR
- S
- FILES

7. Recipe Variables - Extra Build Information

- EXTRA_OECONF
- EXTRA_OEMAKE
- PACKAGES
- DEFAULT_PREFERENCE

Appendix H. FAQ

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

Poky is a derivative of OpenEmbedded [http://www.openembedded.org/], a stable, smaller subset focused on the GNOME Mobile environment. Development in Poky is closely tied to OpenEmbedded with features being merged regularly between the two for mutual benefit.

H.2. 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 a Buildbot 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 [http://vernier.com/labquest/] is using Poky (for more about the Labquest see the case study at OpenedHand). There are a number of pre-production devices using Poky and we will announce those as soon as they are released.

H.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, ipkg, Debian package, or RPM. The package 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 [http://www.gnome.org/mobile/] is a subset of the GNOME platform targeted at mobile and embedded devices. The the main difference between GNOME Mobile and standard GNOME is that desktop-orientated libraries have been removed, along with deprecated libraries, creating a much smaller footprint.

H.9.
How do I make Poky work in RHEL/CentOS?

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

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

On top of those the following external packages are needed:

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

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

H.10.
I see lots of 404 responses for files on http://folks.o-hand.com/~richard/poky/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.11. 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.

Appendix I. Contributing to Poky

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.

2. Bugtracker

Problems with Poky should be reported in the bug tracker [http://bugzilla.o-hand.com/].

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 [http://lists.o-hand.com/poky/].

4. IRC

Join #poky on freenode.

5. Links

- The Poky website [http://pokylinux.org]
- OpenedHand [http://www.openedhand.com/] - The company behind Poky.
- OpenEmbedded [http://www.openembedded.org/] - The upstream generic embedded distribution Poky derives from (and contributes to).
- Bitbake [http://developer.berlios.de/projects/bitbake/] - The tool used to process Poky metadata.
- Bitbake User Manual [http://bitbake.berlios.de/manual/]
- Pimlico [http://pimlico-project.org/] - A suite of lightweight Personal Information Management (PIM) applications designed primarily for handheld and mobile devices.
- QEMU [http://fabrice.bellard.free.fr/qemu/] - An open source machine emulator and virtualizer.

Appendix J. OpenedHand Contact Information

OpenedHand Ltd
Unit R, Homesdale Business Center
216-218 Homesdale Rd
Bromley, BR1 2QZ
England
+44 (0) 208 819 6559
info@openedhand.com

Index