OpenEmbedded User Manual

OpenEmbedded Team

OpenEmbedded User Manual
by OpenEmbedded Team

Table of Contents

Chapter 1. Introduction

1.1. Overview

Like any build tool (make, ant, jam), the OpenEmbedded build tool BitBake controls how to build things and the build dependencies. But unlike single project tools like **make** it is not based on one makefile or a closed set of inter-dependent makefiles, but collects and manages an open set of largely independent build descriptions (package recipes) and builds them in proper order.

To be more precise: OpenEmbedded (http://www.openembedded.org) is a set of metadata used to cross-compile, package and install software packages. OpenEmbedded is being used to build and maintain a number of embedded Linux distributions, including OpenZaurus, Ångström, Familiar and SlugOS.

The primary use-case of OpenEmbedded are:

- Handle cross-compilation.
- Handle inter-package dependencies
- Must be able to emit packages (tar, rpm, deb, ipk)
- Must be able to create images and feeds from packages
- Must be highly configurable to support many machines, distribution and architectures.
- Writing of metadata must be easy and reusable

Together with BitBake (http://bitbake.berlios.de/manual), OpenEmbedded satisfies all these and many more. Flexibility and power have always been the priorities.

1.2. History

OpenEmbedded was invented and founded by the creators of the OpenZaurus project. At this time the project had pushed *buildroot* to its limits. It supported the creation of *ipk* packages, feeds and images and had support for more than one machine. But it was impossible to use different patches, files for different architectures, machines or distributions. To overcome this shortcoming OpenEmbedded was created.

After a few months other projects started using OpenEmbedded and contributing back. On 7 December 2004 Chris Larson split the project into two parts: BitBake, a generic task executor and OpenEmbedded, the metadata for BitBake.

Chapter 2. Getting Started

2.1. OpenEmbedded Directory Structure

Before you begin downloading OpenEmbedded, you need to setup your working environment.

The first step is to decide where on your system you wish to work. This document will use the `$OEBASE` variable to denote the base directory of the OpenEmbedded environment. For example, `$OEBASE` could be `/home/joe/work/oe`.

The base directory of your OpenEmbedded environment (`$OEBASE`) is the location where sources will be checked out (or unpacked). You must choose a location with *no symlinks above it*.

To create the directory structure:

```
$ mkdir -p $OEBASE/build/conf
$ cd $OEBASE
```

The `$OEBASE/build` directory will contain your local configurations and extensions to the OpenEmbedded system which allow you to build your applications and images.

The `$OEBASE` will also contain both of the `bitbake/` and `openembedded/` directories. These will be discussed in Section 2.2 and Section 2.3.

2.2. Getting BitBake

Before using OE, you must first obtain the build tool it needs: bitbake.

It is recommended to run bitbake without installing it, as a sibling directory of `openembedded/` and `build/` directories. Indeed, as bitbake is written in python it does not need compilation for being used. You'll just have to set the `PATH` variable so that the BitBake tools are accessible (see Section 2.4).

2.2.1. Downloading a BitBake release

Releases are available from the berlios project website. The current release series is BitBake *1.8* and the current release is *1.8.18*. To download execute the following commands:

```
$ cd $OEBASE
$ wget http://download.berlios.de/bitbake/bitbake-1.8.18.tar.gz
$ tar -xvzf bitbake-1.8.18.tar.gz
```

```
$ mv bitbake-1.8.18 bitbake
```

BitBake is now downloaded and the `$OEBASE` directory will contain a `bitbake/` subdirectory.

2.3. Getting OpenEmbedded

Note: Once upon a time OpenEmbedded used Monotone for version control. If you have an OE Monotone repository on your computer, you should replace it with the Git repository.

The OpenEmbedded metadata has a high rate of development, so it's a good idea to stay up to date. You'll need Git to get the metadata and stay up to date. Git is available in most distributions and has binaries at Git homepage (http://git-scm.com/).

2.3.1. Checking Out OpenEmbedded With Git

Once you have installed Git, checkout the OpenEmbedded repository:

```
$ cd $OEBASE
$ git clone git://git.openembedded.org/openembedded
```

The `$OEBASE/openembedded/` directory should now exist.

2.3.2. Updating OpenEmbedded

The `org.openembedded.dev` branch of OpenEmbedded is updated very frequently (as much as several times an hour). The distro branches are not updated as much but still fairly often. It seems good practice to update your OpenEmbedded tree at least daily. To do this, run:

```
$ cd $OEBASE/openembedded
$ git pull
```

2.3.3. Changing Branches

Working with multiple branches is very easy to do with Git. The OpenEmbedded repository holds many branches. To list all branches, use this command:

```
$ git branch -a
```

Branch names that begin with `origin/` denote branches that exist on the remote server. The name with a * in front of it is the branch currently checked out. If you want to work with a remote branch, you must first create a local copy of it. The following command will create a local copy of a remote branch:

```
$ git branch <local_name> <remote_name>
```

To change branches, use this command:

```
$ git checkout <branch_name>
```

There are more complicated branch operations that can be done with git, but those are beyond the scope of this document.

2.4. Configuring OpenEmbedded

At this point, your `$OEBASE/` directory should contain at least the following subdirectories:

- `build/`
- `bitbake/`
- `openembedded/`

2.4.1. Environment Setup

There are a few environment variables that you will need to set before you can build software for OpenEmbedded using BitBake. You will need to set these variables every time you open a terminal for development. You can automate this in `~/.profile`, `/etc/profile`, or perhaps use a script to set the necessary variables for using BitBake.

Since the path to your OpenEmbedded installation will be used in many places, setting it in your environment will allow you to use the `$OEBASE` variable in all paths and make it easier to change in the future should the need arise. To set `$OEBASE` if you use a Bourne like shell [1], do this:

```
$ export OEBASE=/path/to/your/oe/installation
```

If you followed the recommendation to use BitBake from svn, you will need to add the path to the BitBake executable to your `PATH` environment variable like this:

```
$ export PATH=$OEBASE/bitbake/bin:$PATH
```

In order for bitbake to find the configuration files for OpenEmbedded, you will need to set the BBPATH variable.

```
$ export BBPATH=$OEBASE/build:$OEBASE/openembedded
```

Finally, if you wish to allow BitBake to inherit the $OEBASE variable from the environment, you will need to set the BB_ENV_EXTRAWHITE variable:

```
$ export BB_ENV_EXTRAWHITE="OEBASE"
```

Note the absence of the "$" character which implies that you are setting BB_ENV_EXTRAWHITE to the variable name, not the variable value.

2.4.2. Local Configuration

It is now time to create your local configuration. While you could copy the default local.conf.sample like this:

```
$ cd $OEBASE
$ cp openembedded/conf/local.conf.sample build/conf/local.conf
$ vi build/conf/local.conf
```

It is actually recommended to start smaller and keep local.conf.sample in the background. Add entries from there step-by-step as you understand and need them. Please, do not just edit build/conf/local.conf.sample but actually *READ* it (read it and then edit it).

For building an org.openembedded.dev branch, in your local.conf file, you should have at least the following three entries: BBFILES, DISTRO and MACHINE. For example, consider the following minimal local.conf file for the Ångström distribution and the Openmoko gta01 machine:

```
BBFILES = "${OEBASE}/openembedded/recipes/*/*.bb"
DISTRO = "angstrom-2008.1"
MACHINE = "om-gta01"
```

2.5. Building Software

The primary interface to the build system is the **bitbake** command (see the BitBake users manual (http://bitbake.berlios.de/manual/)). BitBake will download and patch files from the internet, so it helps if you are on a well connected machine.

Note that you should issue all BitBake commands from inside of the `build/` directory, or you should override `TMPDIR` in your `$OEBASE/build/conf/local.conf` to point elsewhere (by default it goes to `tmp/` relative to the directory you run **bitbake** commands in).

> **Note:** BitBake might complain that there is a problem with the setting in `/proc/sys/vm/mmap_min_addr`, which needs to be set to zero. You can set it by doing the following as root:
>
> ```
> # echo 0 > /proc/sys/vm/mmap_min_addr
> ```
>
> Note that you can not use a text editor to do this since files in `/proc` are not real files. Also note that this above change will be lost when you reboot your system. To have the change made automatically when the system boots, some systems provide a `/etc/sysctl.conf` file. Add the following line to that file:
>
> ```
> vm.mmap_min_addr=0
> ```
>
> If your system does not provide the `/etc/sysctl.conf` mechanism, you can try adding the above **echo** command line to your `/etc/rc.local`. But that's not all. On some systems (such as Fedora 11), changing that kernel setting will cause an SELinux violation if you're running SELinux in enforcing mode. If that's the case, you can either disable SELinux or run:
>
> ```
> $ setsebool -P allow_unconfirmed_mmap_low 1
> ```

Once BitBake and OpenEmbedded are set up and configured, you can build software and images like this:

```
$ bitbake <recipe_name>
```

A recipe name corresponds to a BitBake `.bb` file. A BitBake file is a logical unit of tasks to be executed. Normally this is a package to be built. Inter-recipe dependencies are obeyed. The recipes are located by BitBake via the `BBFILES` variable (set in your `$OEBASE/build/conf/local.conf`), which is a space separated list of `.bb` files, and does handle wildcards.

To build a single package, bypassing the long parse step (and therefore its dependencies -- use with care):

```
$ bitbake -b $OEBASE/openembedded/recipes/blah/blah.bb
```

There are a few groups of special recipes located in subdirectories of the `$OEBASE/openembedded/recipes/` directory. These groups are:

`tasks/`

A collection of meta-packages that depend on real packages to make managing package sets easier.

`meta/`

A collection of useful meta tasks and recipes that don't fit in a general category.

`images/`

A collection of image targets that depend on packages that will be installed into an image which can be put on the target system.

2.5.1. Useful Target Recipes

Although BitBake can build individual packages, it is often more useful to build a set of packages and combine them into an image. The following recipe names are commonly used to that effect.

2.5.1.1. Images

`helloworld-image`

Builds an image, that if used as a root filesystem, will start a static executable that prints hello world then loops infinitely. Can be used to test the Linux boot procedure into user space (init).

`bootstrap-image`

Build image contains task-base packages.

`base-image`

Build an image that is the smallest possible image which allows for ssh access and the ability to install additional packages using ipkg.

`console-image`

Build an image without the X11, gtk+, or qt windowing libraries.

`x11-image`

Builds an image with X11.

`beagleboard-demo-image`

Builds the Ångström distribution like Koen proposed.

`opie-image`

Build image based on the Open Palmtop Integrated Environment (http://opie.handhelds.org/) (OPIE). OPIE is a completely Open Source based graphical user environment and suite of applications for small form-factor devices, such as PDAs, running Linux.

`opie-kdepim-image`

Build image based on the OPIE and full featured KDE-based PIM (pi-sync, ko/pi, ka/pi, etc).

`pivotboot-image`

Build image that is necessary to flash a Sharp SL C3000, Zaurus. It pivots after booting from the NAND and finalizes the install to the HD during the first boot.

`twin-image`

A image with task-base plus a couple of editors, nano and vim (why two?), and a mail reader, mutt.

`uml-image`

A root image for user-mode-linux. Includes task-base, and parts of opie.

`gpe-image`

Build a GPE Palmtop Environment (http://opie.handhelds.org/) based kernel and rootfs. The GPE provides a user interface environment for palmtop/handheld computers running the GNU/Linux or any other UNIX-like operating system.

2.5.1.2. Tasks

`task-base`

Build a kernel and core packages for a basic installation. You won't be able to do much more than ssh to the machine if this is all that is installed.

`task-dvb`

Meta-package for DVB application (DVB = Digital Video Broadcasting).

`task-python-everything`

All of python.

`task-sdk-native`

Meta-package for native (on-device) SDK. i.e. libc, gcc, make, etc.

2.5.1.3. Meta

`meta-opie`

Build all OPIE related packages and some more for OPIE based usage.

`meta-gpe`

Basic packages to go with gpe-image.

2.5.1.4. Other

`helloworld`

Builds a static executable that prints hello world then loops infinitely.

`world`

Build everything. This takes a long time, a lot of network bandwidth, and a lot of disc space. Can also break your toolchain.

`package-index`

Target to update the "feed" files to reflect the current set of .ipk's that exist in the deploy directory. Commonly used after building some packages individually to update the feed and allow them to be installed via a package manager or the ipkg command line tools.

`virtual/kernel`

Builds the appropriate kernel for your device.

`virtual/bootloader`

Builds the appropriate bootloader for your device.

Notes

1. If you use a CSH like shell (e.g. on a FreeBSD system), you will set environment variables like this:

```
$ setenv VAR_NAME "VAR_VALUE"
```

Chapter 3. Metadata

3.1. File Layout

The OpenEmbedded directory, `$OEBASE/openembedded/`, has seven directories, three of which hold BitBake metadata.

`classes/`

Contains BitBake `.bbclass` files. These classes can be inherited by other BitBake files. Every BitBake `.bb` file automatically inherits the `base.bbclass`. `BBPATH` is used to find the `.bbclass` files.

`conf/`

Contains the configuration files for OpenEmbedded. The `bitbake.conf` is read when BitBake is started and this will include the `local.conf`, the machine and distribution configuration files, among others. These files will be located using the `BBPATH` environment variable as a search path.

`contrib/`

Contains miscellaneous scripts that do not belong in the other directories.

`docs/`

Contains the source for the user manual and other documentation files.

`files/`

Contains setup tables for populating the `/dev` directory of various target images.

`recipes/`

Contains all of the BitBake `.bb` files. There is a subdirectory for each task or application and within that subdirectory is a BitBake `.bb` file for each supported version of an application or task.

`site/`

Contains site configuration files for the autoconf/automake system.

3.2. Syntax

OpenEmbedded has files ending with *.conf*, *.inc*, *.bb* and *.bbclass*. The syntax and semantics of these files are best described in the BitBake manual (http://bitbake.berlios.de/manual).

3.3. Classes

OpenEmbedded provides special BitBake classes to ease compiling, packaging and other things. FIXME.

3.4. Writing Meta Data (Adding packages)

This page will guide you through the effort of writing a .bb file or *recipe* in BitBake speak.

Let's start with the easy stuff, like the package description, license, etc:

```
DESCRIPTION = "My first application, a really cool app containing lots of foo and bar"
LICENSE = "GPLv2"
HOMEPAGE = "http://www.host.com/foo/"
```

The description and license fields are mandatory, so better check them twice.

The next step is to specify what the package needs to build and run, the so called *dependencies*:

```
DEPENDS = "gtk+"
RDEPENDS = "cool-ttf-fonts"
```

The package needs gtk+ to build ('DEPENDS') and requires the 'cool-ttf-fonts' package to run ('RDEPENDS'). OE will add run-time dependencies on libraries on its own via the so called *shlibs*-code, but you need to specify everything else by yourself, which in this case is the 'cool-ttf-fonts' package.

After entering all this OE will know what to build before trying to build your application, but it doesn't know where to get it yet. So let's add the source location:

```
SRC_URI = "http://www.host.com/foo/files/${P}.tar.bz2;md5sum=yoursum"
```

This will tell the fetcher where to download the sources from and it will check the integrity using md5sum if you provided the appropriate *yoursum*. You can make one by doing

```
md5sum foo-1.9.tar.bz2
```

and replacing *yoursum* with the md5sum on your screen. A typical md5sum will look like this:

```
a6434b0fc8a54c3dec3d6875bf3be868
```

Notice the *${P}* variable holds the package name (*${PN}* in BitBake speak) and the package version (*${PV}* in BitBake speak). It's a short way of writing *${PN}-${PV}*. Using this notation means you can copy the recipe when a new version is released without having to alter the contents. You do need to check if everything is still correct, because new versions mean new bugs.

Before we can move to the actual building we need to find out which build system the package is using. If we're lucky, we see a *configure* file in the build tree this is an indicator that we can *inherit autotools* if we see a *.pro* file, it might be qmake, which needs *inherit qmake*. Virtually all gtk apps use autotools:

```
inherit autotools pkgconfig
```

We are in luck! The package is a well-behaved application using autotools and pkgconfig to configure and build it self.

Lets start the build:

```
bitbake foo
```

Depending on what you have built before and the speed of your computer this can take a few seconds to a few hours, so be prepared.

.... some time goes by .....

Your screen should now have something like this on it:

```
NOTE: package foo-1.9-r0: task do_build: completed
NOTE: package foo-1.9: completed
NOTE: build 200605052219: completed
```

All looks well, but wait, let's scroll up:

```
NOTE: the following files where installed but not shipped:
    /usr/weirdpath/importantfile.foo
```

OE has a standard list of paths which need to be included, but it can't know everything, so we have to tell OE to include that file as well:

```
FILES_${PN} += "/usr/weirdpath/importantfile.foo"
```

It's important to use += so it will get appended to the standard file-list, not replace the standard one.

Chapter 4. Special features

4.1. Debian package naming

```
INHERIT += "debian"
```

Placing the above line into your *${DISTRO}.conf* or *local.conf* will trigger renaming of packages if they only ship one library. Imagine a package where the package name (**PN**) is foo and this packages ships a file named **libfoo.so.1.2.3**. Now this package will be renamed to **libfoo1** to follow the Debian package naming policy.

4.2. Shared Library handling (shlibs)

Run-time Dependencies (**RDEPENDS**) will be added when packaging the software. They should only contain the minimal dependencies to run the program. OpenEmbedded will analyze each packaged binary and search for **SO_NEEDED** libraries. These libraries are absolutely required by the program so OpenEmbedded will search for packages that install these libraries. These packages are automatically added to the **RDEPENDS**. As a packager you don't need to worry about shared libraries anymore they will be added automatically.

* *NOTE: This does not apply to plug-ins used by the program.*

4.3. BitBake Collections

This section is a stub, help us by expanding it

```
BBFILES := "${OEDIR}/openembedded/recipes/*/*.bb ${LOCALDIR}/recipes/*/*.bb"
BBFILE_COLLECTIONS = "upstream local"
BBFILE_PATTERN_upstream = "^${OEDIR}/openembedded/recipes/"
BBFILE_PATTERN_local = "^${LOCALDIR}/recipes/"
BBFILE_PRIORITY_upstream = "5"
BBFILE_PRIORITY_local = "10"
```

4.4. Task-base

Task-base is new way of creating basic root filesystems. Instead of having each machine setting a ton of duplicate variables, this allows a machine to specify its features and **task-base** builds it a customized

package based on what the machine needs along with what the distro supports.

To illustrate, the distro config file can say:

```
DISTRO_FEATURES = "nfs smbfs ipsec wifi ppp alsa bluetooth ext2 irda pcmcia usbgadget usbho
```

and the machine config:

```
MACHINE_FEATURES = "kernel26 apm alsa pcmcia bluetooth irda usbgadget"
```

and the resulting **task-base** would support pcmcia but not usbhost.

Task-base details exactly which options are either machine or distro settings (or need to be in both). Machine options are meant to reflect capabilities of the machine, distro options list things distribution maintainers might want to add or remove from their distros images.

4.5. Overrides

This section is a stub, help us by expanding it

Chapter 5. Common Use-cases/tasks

5.1. Creating a new Distribution

Creating a new distribution is not complicated, however we urge you to try existing distributions first, because it's also very easy to do wrong. The config needs to be created in $OEBASE/openembedded/conf/distro directory. So what has to be inside?

- **DISTRO_VERSION** so users will know which version of the distribution they are using.
- **DISTRO_TYPE** (release/debug) variable is used in some recipes to enable/disable some features - for example kernel output on screen for "debug" builds.
- Type of libc used: will it be glibc (**TARGET_OS** = "linux") or uclibc (**TARGET_OS** = "linux-uclibc")?
- Toolchain versions - for example gcc 3.4.4 based distro will have:

```
PREFERRED_PROVIDERS += " virtual/${TARGET_PREFIX}gcc-initial:gcc-cross-initial"
PREFERRED_PROVIDERS += " virtual/${TARGET_PREFIX}gcc:gcc-cross"
PREFERRED_PROVIDERS += " virtual/${TARGET_PREFIX}g++:gcc-cross"

PREFERRED_VERSION_binutils = "2.16"
PREFERRED_VERSION_binutils-cross = "2.16"

PREFERRED_VERSION_gcc = "3.4.4"
PREFERRED_VERSION_gcc-cross = "3.4.4"
PREFERRED_VERSION_gcc-initial-cross = "3.4.4"
```

- **DISTRO_FEATURES** which describe which features distro has. More about it in task-base section.
- Versions of kernels used for supported devices:

```
PREFERRED_VERSION_linux-omap1_omap5912osk ?= "2.6.18+git"
PREFERRED_VERSION_linux-openzaurus ?= "2.6.17"
```

5.2. Adding a new Machine

To be able to build for a device OpenEmbedded has to know about it, so a machine config file needs to be written. All of the machine configs are stored in $OEBASE/openembedded/conf/machine/ directory.

As usual some variables are required:

- **TARGET_ARCH** describes which CPU architecture the machine uses.

- **MACHINE_FEATURES** which describes which features the device has. More about it in task-base section.
- **PREFERRED_PROVIDER_virtual/kernel** has to point to the proper kernel recipe for this machine.

There are also some optional variables that can be defined:

- **MACHINE_OVERRIDES** lists additional items to add to the **OVERRIDES** variable, between the **DISTRO** and the **MACHINE**. This is utilized to add overrides which are less specific than the machine, but are nonetheless related to it, allowing us to define variables a certain way for a group of machines, rather than for each individual one. As an example, this variable may be used by the distribution to add **SOC_FAMILY** or **MACHINE_CLASS**.

  Note that this variable is space separated, and should always be manipulated with +=, to ensure it's built up incrementally, and no additions are lost.

- **SOC_FAMILY** describes a family of processors that all share common features such as kernel versions, bootloaders, etc. This is used to allow overrides for a whole set of devices rather than per machine overrides being used. The use of SOC_FAMILY as an override is currently a distribution or local setting.

  NOTE: SOC_FAMILY is different than MACHINE_CLASS in that MACHINE_CLASS is intended to specify a grouping of devices that may have different processors but share common features. For example all OMAP3 devices can be described using the SOC_FAMILY "omap3" and this value can be used in overrides to prevent requiring multiple machine specific overrides. MACHINE_CLASS might be used to describe a class of devices such as a cell phone in which the processor may be different but the features such as touchscreen, GPS, modem, etc are the same.

Next the kernel recipe needs to be added if it doesn't already exist.

5.3. Adding a new Package

This section is a stub, help us by expanding it. Learn by example, go through the recipes that are already there and mimic them to do what you want.

5.3.1. building from unstable source code

Building against the latest, bleeding-edge source has some intricacies of its own. For one, it is desirable to pin down a code revision that is known to build to prevent random breakage in OE at the most

inopportune time for all OE users. Here is how to do that properly.

- for svn: add 'PV = "1.1+svnr${SRCPV}"' to your bb file.
- for git: add 'PV = "1.1+gitr${SRCPV}"' to your bb file.
- for cvs: add 'PV = "1.1+cvs${SRCPV}"' to your bb file.

Accompany with stable SRCREV for your package directly in the package recipe.

If you really absolutely have to follow the latest commits, you can do that by adding 'SRCREV_pn-linux-davinci ?= ${AUTOREV}' to your local.conf, for example. In this case, you'd build against the most recent and unstable source for the pn-linux-davinci package.

5.4. Creating your own image

Creating own image is easy - only few variables need to be set:

- **IMAGE_BASENAME** to give a name for your own image
- **PACKAGE_INSTALL** to give a list of packages to install into the image
- **RDEPENDS** to give a list of recipes which are needed to be built to create this image
- **IMAGE_LINGUAS** is an optional list of languages which has to be installed into the image

Then add the *image* class using:

```
inherit image
```

And the image recipe is ready for usage.

5.5. Using a prebuilt toolchain to create your packages

TODO: You want to use external-toolchain. Setting PREFERRED_PROVIDER for the toolchain to that + environment variables + toolchain layout. Please someone write the documentation for that.

5.6. Using a new package format

This section is a stub, help us by expanding it

5.7. Creating Software Development Kits (SDKs)

5.7.1. What is provided by a SDK

The Software Development Kit (SDK) should be easy to install and enable your user-base to create binaries and libraries that work on the target hardware.

To accomplish this goal OpenEmbedded SDKs contain tools for the host and tools for the target hardware. Among these tools is a cross compiler, libraries and header files for additional dependencies, pkg-config files to allow buildsystems to easily find the dependencies, a file with results for autoconf and a script that can be sourced to setup the environment.

5.7.2. Creating a SDK with your libraries pre-installed

5.7.2.1. Preparing the host side

Your SDK might need utilities that will run on the host. These could include scripts, buildsystem software like cmake, or an emulator like qemu. For these dependencies it is imported that they *inherit sdk* and by convention end with *-sdk* in the **PN**.

A new task should be created that will assure that all host utilities will be installed. Place a file called `task-YOUR-toolchain-host.bb` in the `recipes/tasks` directory and place the following content in it:

```
require task-sdk-host.bb
DESCRIPTION = "Host packages for YOUR SDK"
LICENSE = "MIT"
ALLOW_EMPTY = "1"
RDEPENDS_${PN} += "YOUR-DEPENDENCY-sdk"
```

5.7.2.2. Preparing the target side

Your SDK should provide your user with header files and libraries he will need when doing application development. In OpenEmbedded the **${PN}-dev** is providing the header files, pkg-config files and symbolic links to libraries to allow using the library. The SDK should install these development packages to the SDK.

To install the development packages you will need to create a new task. Create a new file `task-YOUR-toolchain-target.bb` in the `recipes/tasks` directory and place the following content in it:

```
DESCRIPTION = "Target package for YOUR SDK"
LICENSE = "MIT"
ALLOW_EMPTY = "1"

PR = "r0"

RDEPENDS_${PN} += "\
        task-sdk-bare \
        your-lib-dev \
        your-data
        "
```

5.7.2.3. Putting it together

In the previous two sections we have prepared the host and target side. One thing that is missing is combining the two newly created tasks and actually creating the SDK. This is what we are going to do now.

Create `meta-toolchain-YOU.bb` in the `recipes/meta` directory and place the following content in it:

```
PR = "r0"
TOOLCHAIN_TARGET_TASK = "task-YOUR-toolchain-target"
TOOLCHAIN_HOST_TASK = "task-YOUR-toolchain-host"

require meta-toolchain.bb
SDK_SUFFIX = "toolchain-YOUR"
```

Using **bitbake meta-toolchain-YOU** the SDK creation should be started and you should find a `sdk` directory inside your deploy directory with a SDK waiting for you. With the above command you still need to have OE configured with your `conf/local.conf` to select the machine and distribution you are targeting.

> **Note:** SDK creation currently does not work with the *DISTRO* set to *micro*.

> **Note:** If the environment-setup script packaged in the SDK should require more environment look at the `meta-toolchain-qte.bb` to accomplish this.

5.8. Creating and Using a Qt Embedded SDK

5.8.1. Creating the SDK

The SDK should contain a build of Qt Embedded, but also optional dependencies like directFB, glib-2.0, gstreamer-0.10, tslib and more esoteric dependencies like mysql and postgres. This allows developers to simply start developing using Qt and enables system integrators to easily recompile Qt and base libraries without tracking down extra dependencies.

OpenEmbedded provides an easy way to create a Qt Embedded SDK. In `recipes/tasks/task-qte-toolchain-host.bb` host tools like moc, uic, rcc, qmake will get installed and in `recipes/tasks/task-qte-toolchain-target.bb` the Qt4 header files and libraries will be installed.

To build the SDK, setup OpenEmbedded in the usual way by picking a *DISTRO* and *MACHINE*. Issue the below command and after the operation finished you should find a SDK in the deployment directory.

```
$ bitbake meta-toolchain-qte
```

Note: The deployment directory depends on the distribution and used C library. In the case of Angstrom and glibc it is located in `tmp/deploy/glibc/sdk`.

Note: Change `qt4-embedded.inc` and `qt4.inc` for using different Qt configuration flags. This might include a custom qconfig.h to produce a reduced size build.

Note: When distributing the SDK make sure to include a written offer to provide the sourcecode of GPL licensed applications or provide parts of the `sources` folder. The `sources` folder is located right next to the `sdk` one.

5.8.2. Using the Qt Embedded SDK

In this example we are assuming that the target hardware is an armv5t system and the SDK targets the Angstrom Distribution. You should start by downloading the SDK and untar it to the root folder (`/`). Once this operation is finished you will find a new directory `/usr/local/angstrom/arm/` and it contains the `environment-setup` file to setup the *QMAKESPEC* and various other paths.

```
Untar the SDK once
```

```
$ tar -C / -xjf angstrom-armv5te-linux-gnueabi-toolchain-qte.tar.bz2

Before using it source the environment
$ . /usr/local/angstrom/arm/environment-setup

Use qmake2 to build software for the target
$ qmake2
```

Creating and building a simple example. We will create a simple Qt Embedded application and use **qmake2** and **make** to cross compile.

```
$ . /usr/local/angstrom/arm/environment-setup
$ cd $HOME
$ mkdir qte-example
$ cd qte-example

$ echo "TEMPLATE=app
SOURCES=main.cpp
" > qte-example.pro

$ echo '#include <QApplication>
#include <QPushButton>

int main(int argc, char** argv) {
    QApplication app(argc, argv);

    QPushButton btn("Hello World");
    btn.show();
    btn.showMaximized();

    return app.exec();
}
' > main.cpp

$ qmake2
$ make
```

Chapter 6. Comparing

6.1. buildroot

Writing of BitBake recipes is easier and more intuitive than writing Makefiles while providing higher flexibility. This allows you to tweak specific recipes for your very special needs and to add new recipes quickly. You can build toolchains, Software Distribution Kits (SDKs), complete Distributions or just single packages. The flexibility of OpenEmbedded allows you to reuse recipes for many different purposes. OpenEmbedded provides everything buildroot will be able to provide. But in contrast to buildroot OpenEmbedded will allow you to achieve what you really want to achieve. You can add new package formats, new filesystems, new output formats easily. OpenEmbedded will suit your need.

6.2. crosstool

Crosstool allows the creation of toolchains. It can only create the initial toolchain for you. It will not compile other needed libraries or applications for you, it will not be able to track dependencies or package them properly. OpenEmbedded supports all configurations crosstool supports. You can start by creating toolchains with OpenEmbedded, then as your needs grow create a more complete SDK from already present base libraries and applications and if you recognize you need to have packages for the target you have them almost built already.

6.3. handmade

Cross-compilation is a tough business. It is not that cross-compiling is hard itself but many people misuse the buildsystem they use to build their software. This will lead to a variety of issues you can run into. This can be failing tests on configuration because of executing cross compiled binaries or crashes at run-time due to wrong sizes of basic types. When utilizing OpenEmbedded you avoid searching for patches at many different places and will be able to get things done more quickly. OpenEmbedded allows you to choose from a pool of ready to use software packages.

OpenEmbedded will create complete flashable images using different output formats and filesystems. This allows you to create complete and specialized distributions easily.

Chapter 7. Using bitbake and OpenEmbedded

7.1. Introduction

If you're reading this manual you probably already have some idea of what OpenEmbedded is all about, which is taking a lot of software and creating something that you can run on another device. This involves downloading some source code, compiling it, creating packages (like .deb or .rpm) and/or creating boot images that can be written to flash on the device. The difficulties of cross-compiling and the variety of devices which can be supported lead to a lot more complexity in an OpenEmbedded based distribution than you'd find in a typical desktop distribution (where which cross-compiling isn't needed).

A major part of OpenEmbedded deals with compiling source code for various projects. For each project this generally requires the same basic set of tasks:

1. Download the source code, and any supporting files (such as initscripts);
2. Extract the source code and apply any patches that might be wanted;
3. Configure the software if needed (such as is done by running the configure script);
4. Compile everything;
5. Package up all the files into some package format, like .deb or .rpm or .ipk, ready for installation.

There's nothing particular unusual about this process when building on the machine the package is to be installed on. What makes this difficult is:

1. Cross-compiling: cross-compiling is difficult, and lots of software has no support for cross-compiling - all packages included in OE are cross-compiled;
2. Target and host are different: This means you can't compile up a program and then run it - it's compiled to run on the target system, not on the system compiling it. Lots of software tries to build and run little helper and/or test applications and this won't work when cross-compiling.
3. Tool chains (compiler, linker etc) are often difficult to compile. Cross tool chains are even more difficult. Typically you'd go out and download a tool chain made by someone else - but not when you're using OE. In OE the entire toolchain is built as part of the process. This may make things take longer initially and may make it more difficult to get started but makes it easier to apply patches and test out changes to the tool chain.

Of course there's a lot more to OE then just compiling packages though. Some of the features that OE supports includes:

- Support for both glibc and uclibc;
- Support for building for multiple target devices from the one code base;
- Automatically building anything that is required for the package to compile and/or run (build and run time dependencies);

- Creation of flash and disk images of any one of a number of types (jffs2, ext2.gz, squashfs etc) for booting directly on the target device;
- Support for various packaging formats;
- Automatic building all of the cross-compiling tools you'll need;
- Support for "native" packages that are built for the host computer and not for the target and used to help during the build process;

The rest of this chapter assumes you have mastered the Getting Start guides to OpenEmbedded (see the OpenEmbedded web site for details), and therefore have an appropriately configured setup and that you have managed to actually build the cross-compilers for your target. This section talks you through some of the background on what is happening with the aim of helping you understand how to debug and develop within OpenEmbedded.

You'll also note a lot of references to variables that define specific directories or change the behaviour of some part of the build process. You should refer to Recipes chapter for full details on these variables.

7.2. Configuration

Configuration covers basic items such as where the various files can be found and where output should be placed to more specific items such as which hardware is being targeted and what features you want to have included in the final image. The main configuration areas in OE are:

conf/machine

This directory contains machine configuration information. For each physical device a configuration file is required in this directory that describes various aspects of the device, such as architecture of the device, hardware features of the device (does it have usb? a keyboard? etc), the type of flash or disk images needed for the device, the serial console settings (if any) etc. If you are adding support for a new device you would need to create a machine configuration in this directory for the device.

conf/distro

This directory contains distribution related files. A distribution decides how various activities are handled in the final image, such as how networking is configured, if usb devices will be supported, what packaging system is used, which libc is used etc.

conf/bitbake.conf

This is the main bitbake configuration file. This file is not to be edited but it is useful to look at it since it declares a larger number of the predefined variables used by OE and controls a lot of the base functionality provided by OE.

conf/local.conf

This is the end-user specific configuration. This file needs to be copied and edited and is used to specify the various working directories, the machine to build for and the distribution to use.

7.3. Work space

Let's start out by taking a look at a typical working area. Note that this may not be exactly what you see - there are a lot of options that can effect exactly how things are done, but it gives us a pretty good idea of whats going on. What we are looking at here is the tmp directory (as specified by TMPDIR in your local.conf):

```
$ find tmp -maxdepth 2 -type d
tmp
tmp/stamps
tmp/cross
tmp/cross/bin
tmp/cross/libexec
tmp/cross/lib
tmp/cross/share
tmp/cross/sh4-linux
tmp/cache
tmp/cache/titan
tmp/work
tmp/work/busybox-1.2.1-r13
tmp/work/libice-1_1.0.3-r0
tmp/work/arpwatch-2.1a15-r2
...
tmp/rootfs
tmp/rootfs/bin
tmp/rootfs/usr
tmp/rootfs/media
tmp/rootfs/dev
tmp/rootfs/var
tmp/rootfs/lib
tmp/rootfs/sbin
tmp/rootfs/mnt
tmp/rootfs/boot
tmp/rootfs/sys
tmp/rootfs/proc
tmp/rootfs/etc
tmp/rootfs/home
tmp/rootfs/tmp
tmp/sysroots
tmp/pkgdata
tmp/deploy
tmp/deploy/addons
tmp/deploy/ipk
tmp/deploy/sources
tmp/deploy/images
```

The various top level directories under tmp include:

stamps

Nothing of interest to users in here. These time stamps are used by bitbake to keep track of what tasks it has completed and what tasks it still has outstanding. This is how it knows that certain actions have been completed and it doesn't need to do them again.

cross

Contains the cross-compiler toolchain. That is the gcc and binutils that run on the host system but produce output for the target system.

cache

Nothing of interest to users in here. This contains the bitbake parse cache and is used to avoid the need to parse all of the recipes each time bitbake is run. This makes bitbake a lot faster on the 2nd and subsequent runs.

work

The work directory. This is the directory in which all packages are built - this is where the source code is extract, patches applied, software configure, compiled, installed and package. This is where you'll spend most of you time looking when working in OE.

rootfs

The generated root filesystem image for your target device. This is the contents of the root filesystem (NOTE: fakeroot means it doesn't have the correct device special nodes and permissions to use directly).

sysroots

Contains the staging area, which is used to store natively compiled tools and and libraries and headers for the target that are required for building other software.

deploy

Contains the final output from OE. This includes the installation packages (typically .ipkg packages) and flash and/or disk images. This is where you go to get the final product.

When people refer to the *"tmp directory"* this is the directory they are talking about.

To perform a complete rebuild from scratch you would usually rename or delete tmp and then restart your build. I recommend keeping one old version of tmp around to use for comparison if something goes wrong with your new build. For example:

```
$ rm -fr tmp.OLD
$ mv tmp tmp.OLD
$ bitbake bootstrap-image
```

7.3.1. work directory (tmp/work)

The work directory is where all source code is unpacked into, where source is configured, compiled and packaged. In other words this is where all the action happens. Each bitbake recipe will produce a corresponding subdirectory in the work directory. The subdirectory name will contain the recipe name, version and the release number (as defined by the PR variable within the recipe).

Here's an example of a few of the subdirectories under the work directory:

```
$ find tmp/work -maxdepth 1 -type d | head -4
tmp/work
tmp/work/busybox-1.2.1-r13
tmp/work/libice-1_1.0.3-r0
tmp/work/arpwatch-2.1a15-r2
```

You can see the first three (of several hundred) recipes here and they are for release 13 of busybox 1.2.1, release 0 of libice 1.1.0.3 and release 2 of arpwatch 2.1a15. It's also possible that you may just have a sub directory for your targets architecture and operating system in which case these directories will be in that additional subdirectory, as shown here:

```
$ find tmp/work -maxdepth 2 -type d | head -4
tmp/work
tmp/work/sh4-linux
tmp/work/sh4-linux/busybox-1.2.1-r13
tmp/work/sh4-linux/libice-1_1.0.3-r0
tmp/work/sh4-linux/arpwatch-2.1a15-r2
```

The **sh4-linux** directory in the above example is a combination of the target architecture (sh4) and operating system (linux). This subdirectory has been added by the use of one of OpenEmbedded's many features. In this case it's the *multimachine* feature which is used to allow builds for multiple targets within the one work directory and can be enabled on a per distribution basis. This feature enables the sharing of native and architecture neutral packages and building for multiple targets that support the same architecture but require different linux kernels (for example). We'll assume multimachine isn't being used for the rest of this chapter, just remember to add the extra directory if your distribution is using it.

Using lzo 1.08 as an example we'll examine the contents of the working directory for a typical recipe:

```
$ find tmp/work/lzo-1.08-r14 -maxdepth 1
tmp/work/lzo-1.08-r14
tmp/work/lzo-1.08-r14/temp
tmp/work/lzo-1.08-r14/lzo-1.08
tmp/work/lzo-1.08-r14/install
tmp/work/lzo-1.08-r14/image
```

The directory, **tmp/work/lzo-1.08-r14**, is know as the *"working directory"* for the recipe and is specified via the **WORKDIR** variable in bitbake. You'll sometimes see recipes refer directly to **WORKDIR** and this is the directory they are referencing. The **1.08** is the version of lzo and **r14** is the release number, as defined by the **PR** variable within the recipe.

Under the working directory (**WORKDIR**) there are four subdirectories:

temp

The temp directories contains logs and in some cases scripts that actually implement specific tasks (such as a script to configure or compile the source).

You can look at the logs in this directory to get more information into what happened (or didn't happen). This is usually the first thing to look at when things are going wrong and these usually need to be included when reporting bugs.

The scripts can be used to see what a particular task, such as configure or compile, is trying to do.

lzo-1.08

This is the unpacked source code directory, which was created when the lzo source code was extracted in this directory. The name and format of this directory is therefore dependent on the actual source code packaging. Within recipes this directory is referred to as **S** and is usually expected to be named like this, that is *"<name>-<version>"*. If the source code extracts to somewhere else then that would need to be declared in the recipe by explicitly setting the value of the variable **S** to the appropriate directory.

image

The image directory (or destination directory) is where the software needs to be installed into in order to be packaged. This directory is referred to as **D** in recipes. So instead of installing binaries into **/usr/bin** and libraries into **/usr/lib** for example you would need to install into **${D}/usr/bin** and **${D}/usr/lib** instead. When installed on the target the ${D} will be not be included so they'll end up in the correct place. You definitely don't want files on your host system being replaced by cross-compiled binaries for your target!

install

The install directory is used to split the installed files into separate packages. One subdirectory is created per package to be generated and the files are moved from the image directory (**D**) over to this directory, and into the appropriate package subdirectory, as each packaging instruction is processed. Typically there will be separate documentation (*-doc*), debugging (*-dbg*) and development (*-dev*) packages automatically created. There are variables such as **FILES_** and **PACKAGES** used in recipes which control the separation of various files into individual packages.

So lets show some examples of the useful information you now have access to.

How about checking out what happened during the configuration of lzo? Well that requires checking the log file for configure that is generated in the temp directory:

```
$ less tmp/work/lzo-1.08-r14/temp/log.do_configure.*
...
checking whether ccache sh4-linux-gcc -ml -m4 suffers the -fschedule-insns bug... unknown
checking whether ccache sh4-linux-gcc -ml -m4 suffers the -fstrength-reduce bug... unknown
checking whether ccache sh4-linux-gcc -ml -m4 accepts -fstrict-aliasing... yes
checking the alignment of the assembler... 0
checking whether to build assembler versions... no
configure: creating ./config.status
config.status: creating Makefile
config.status: creating examples/Makefile
config.status: creating include/Makefile
config.status: creating ltest/Makefile
config.status: creating minilzo/Makefile
config.status: creating src/Makefile
config.status: creating tests/Makefile
config.status: creating config.h
config.status: executing depfiles commands
```

Or perhaps you want to see how the files were distributed into individual packages prior to packaging? The install directory is where the files are split into separate packages and so that shows us which files end up where:

```
$ find tmp/work/lzo-1.08-r14/install
tmp/work/lzo-1.08-r14/install
tmp/work/lzo-1.08-r14/install/lzo-doc
tmp/work/lzo-1.08-r14/install/lzo-dbg
tmp/work/lzo-1.08-r14/install/lzo-dbg/usr
tmp/work/lzo-1.08-r14/install/lzo-dbg/usr/lib
tmp/work/lzo-1.08-r14/install/lzo-dbg/usr/lib/.debug
tmp/work/lzo-1.08-r14/install/lzo-dbg/usr/lib/.debug/liblzo.so.1.0.0
tmp/work/lzo-1.08-r14/install/lzo-dev
tmp/work/lzo-1.08-r14/install/lzo-dev/usr
tmp/work/lzo-1.08-r14/install/lzo-dev/usr/include
tmp/work/lzo-1.08-r14/install/lzo-dev/usr/include/lzo2a.h
tmp/work/lzo-1.08-r14/install/lzo-dev/usr/include/lzo1y.h
tmp/work/lzo-1.08-r14/install/lzo-dev/usr/include/lzo1.h
tmp/work/lzo-1.08-r14/install/lzo-dev/usr/include/lzo1b.h
tmp/work/lzo-1.08-r14/install/lzo-dev/usr/include/lzo1f.h
tmp/work/lzo-1.08-r14/install/lzo-dev/usr/include/lzoconf.h
tmp/work/lzo-1.08-r14/install/lzo-dev/usr/include/lzo1x.h
tmp/work/lzo-1.08-r14/install/lzo-dev/usr/include/lzo16bit.h
tmp/work/lzo-1.08-r14/install/lzo-dev/usr/include/lzo1a.h
tmp/work/lzo-1.08-r14/install/lzo-dev/usr/include/lzo1z.h
tmp/work/lzo-1.08-r14/install/lzo-dev/usr/include/lzoutil.h
tmp/work/lzo-1.08-r14/install/lzo-dev/usr/include/lzo1c.h
tmp/work/lzo-1.08-r14/install/lzo-dev/usr/lib
tmp/work/lzo-1.08-r14/install/lzo-dev/usr/lib/liblzo.a
```

```
tmp/work/lzo-1.08-r14/install/lzo-dev/usr/lib/liblzo.so
tmp/work/lzo-1.08-r14/install/lzo-dev/usr/lib/liblzo.la
tmp/work/lzo-1.08-r14/install/lzo.shlibdeps
tmp/work/lzo-1.08-r14/install/lzo-locale
tmp/work/lzo-1.08-r14/install/lzo
tmp/work/lzo-1.08-r14/install/lzo/usr
tmp/work/lzo-1.08-r14/install/lzo/usr/lib
tmp/work/lzo-1.08-r14/install/lzo/usr/lib/liblzo.so.1
tmp/work/lzo-1.08-r14/install/lzo/usr/lib/liblzo.so.1.0.0
```

7.4. Tasks

When you go about building and installing a software package there are a number of tasks that you generally follow with most software packages. You probably need to start out by downloading the source code, then unpacking the source code. Maybe you need to apply some patches for some reason. Then you might run the configure script of the package, perhaps passing it some options to configure it to your liking. Then you might run "make install" to install the software. If you're actually going to make some packages, such as .deb or .rpm, then you'd have additional tasks you'd perform to make them.

You find that building things in OpenEmbedded works in a similar way - there are a number of tasks that are executed in a predefined order for each recipe. Many of the tasks correspond to those listed above like *"download the source"*. In fact you've probably already seen some of the names of these tasks - bitbake displays them as they are processed:

```
$ bitbake lzo
NOTE: Psyco JIT Compiler (http://psyco.sf.net) not available. Install it to increase perfor
NOTE: Handling BitBake files: \ (4541/4541) [100 %]
NOTE: Parsing finished. 4325 cached, 0 parsed, 216 skipped, 0 masked.
NOTE: build 200705041709: started

OE Build Configuration:
BB_VERSION     = "1.8.2"
OE_REVISION    = "<unknown>"
TARGET_ARCH    = "sh4"
TARGET_OS      = "linux"
MACHINE        = "titan"
DISTRO         = "erouter"
DISTRO_VERSION = "0.1-20070504"
TARGET_FPU     = ""

NOTE: Resolving missing task queue dependencies
NOTE: preferred version 2.5 of glibc not available (for item virtual/sh4-linux-libc-for-gcc
NOTE: Preparing Runqueue
NOTE: Executing runqueue
NOTE: Running task 208 of 226 (ID: 11, /home/lenehan/devel/oe/build/titan-glibc-25/recipes/
NOTE: package lzo-1.08: started
```

```
NOTE: package lzo-1.08-r14: task do_fetch: started
NOTE: package lzo-1.08-r14: task do_fetch: completed
NOTE: package lzo-1.08: completed
NOTE: Running task 209 of 226 (ID: 2, /home/lenehan/devel/oe/build/titan-glibc-25/recipes/l
NOTE: package lzo-1.08: started
NOTE: package lzo-1.08-r14: task do_unpack: started
NOTE: Unpacking /home/lenehan/devel/oe/sources/lzo-1.08.tar.gz to /home/lenehan/devel/oe/bu
NOTE: package lzo-1.08-r14: task do_unpack: completed
NOTE: package lzo-1.08: completed
NOTE: Running task 216 of 226 (ID: 3, /home/lenehan/devel/oe/build/titan-glibc-25/recipes/l
NOTE: package lzo-1.08: started
NOTE: package lzo-1.08-r14: task do_patch: started
NOTE: package lzo-1.08-r14: task do_patch: completed
NOTE: package lzo-1.08: completed
NOTE: Running task 217 of 226 (ID: 4, /home/lenehan/devel/oe/build/titan-glibc-25/recipes/l
NOTE: package lzo-1.08: started
NOTE: package lzo-1.08-r14: task do_configure: started
NOTE: package lzo-1.08-r14: task do_configure: completed
NOTE: package lzo-1.08: completed
NOTE: Running task 218 of 226 (ID: 12, /home/lenehan/devel/oe/build/titan-glibc-25/recipes/
NOTE: package lzo-1.08: started
NOTE: package lzo-1.08-r14: task do_qa_configure: started
NOTE: Checking sanity of the config.log file
NOTE: package lzo-1.08-r14: task do_qa_configure: completed
NOTE: package lzo-1.08: completed
NOTE: Running task 219 of 226 (ID: 0, /home/lenehan/devel/oe/build/titan-glibc-25/recipes/l
NOTE: package lzo-1.08: started
NOTE: package lzo-1.08-r14: task do_compile: started
NOTE: package lzo-1.08-r14: task do_compile: completed
NOTE: package lzo-1.08: completed
NOTE: Running task 220 of 226 (ID: 1, /home/lenehan/devel/oe/build/titan-glibc-25/recipes/l
NOTE: package lzo-1.08: started
NOTE: package lzo-1.08-r14: task do_install: started
NOTE: package lzo-1.08-r14: task do_install: completed
NOTE: package lzo-1.08: completed
NOTE: Running task 221 of 226 (ID: 5, /home/lenehan/devel/oe/build/titan-glibc-25/recipes/l
NOTE: package lzo-1.08: started
NOTE: package lzo-1.08-r14: task do_package: started
NOTE: DO PACKAGE QA
NOTE: Checking Package: lzo-dbg
NOTE: Checking Package: lzo
NOTE: Checking Package: lzo-doc
NOTE: Checking Package: lzo-dev
NOTE: Checking Package: lzo-locale
NOTE: DONE with PACKAGE QA
NOTE: package lzo-1.08-r14: task do_package: completed
NOTE: package lzo-1.08: completed
NOTE: Running task 222 of 226 (ID: 8, /home/lenehan/devel/oe/build/titan-glibc-25/recipes/l
NOTE: package lzo-1.08: started
NOTE: package lzo-1.08-r14: task do_package_write: started
Packaged contents of lzo-dbg into /home/lenehan/devel/oe/build/titan-glibc-25/tmp/deploy/ip
Packaged contents of lzo into /home/lenehan/devel/oe/build/titan-glibc-25/tmp/deploy/ipk/sh
NOTE: Not creating empty archive for lzo-doc-1.08-r14
```

```
Packaged contents of lzo-dev into /home/lenehan/devel/oe/build/titan-glibc-25/tmp/deploy/ip
NOTE: Not creating empty archive for lzo-locale-1.08-r14
NOTE: package lzo-1.08-r14: task do_package_write: completed
NOTE: package lzo-1.08: completed
NOTE: Running task 223 of 226 (ID: 6, /home/lenehan/devel/oe/build/titan-glibc-25/recipes/l
NOTE: package lzo-1.08: started
NOTE: package lzo-1.08-r14: task do_populate_sysroot: started
NOTE: package lzo-1.08-r14: task do_populate_sysroot: completed
NOTE: package lzo-1.08: completed
NOTE: Running task 224 of 226 (ID: 9, /home/lenehan/devel/oe/build/titan-glibc-25/recipes/l
NOTE: package lzo-1.08: started
NOTE: package lzo-1.08-r14: task do_qa_staging: started
NOTE: QA checking staging
NOTE: package lzo-1.08-r14: task do_qa_staging: completed
NOTE: package lzo-1.08: completed
NOTE: Running task 225 of 226 (ID: 7, /home/lenehan/devel/oe/build/titan-glibc-25/recipes/l
NOTE: package lzo-1.08: started
NOTE: package lzo-1.08-r14: task do_distribute_sources: started
NOTE: package lzo-1.08-r14: task do_distribute_sources: completed
NOTE: package lzo-1.08: completed
NOTE: Running task 226 of 226 (ID: 10, /home/lenehan/devel/oe/build/titan-glibc-25/recipes/
NOTE: package lzo-1.08: started
NOTE: package lzo-1.08-r14: task do_build: started
NOTE: package lzo-1.08-r14: task do_build: completed
NOTE: package lzo-1.08: completed
NOTE: Tasks Summary: Attempted 226 tasks of which 213 didn't need to be rerun and 0 failed.
NOTE: build 200705041709: completed
```

> **Note:** The output may look different depending on the version of bitbake being used, and some tasks are only run when specific options are enabled in your distribution. The important point to note is that the various tasks are being run and bitbake shows you each time it starts and completes a task.

So there's a set of tasks here which are being run to generate the final packages. And if you'll notice that every recipe runs through the same set of tasks (ok I'll admit that it is possible that some additional tasks could be run for some recipes, but we'll talk about that later). The tasks that you'll need to be most familiar with are:

fetch

> The *fetch* task is responsible for fetching any source code that is required. This means things such as downloading files and checking out from source control repositories such as git or svn.

unpack

> The *unpack* task is responsible for extracting files from archives, such as **.tar.gz**, into the working area and copying any additional files, such as init scripts, into the working area.

patch

The *patch* task is responsible for applying any patches to the unpacked source code

configure

The *configure* task takes care of the configuration of the package. Running a configure script (*"./configure <options>"*) is probably the form of configuration that is most recognized but it's not the only configuration system that exists.

compile

The *compile* task actually compiles the software. This could be as simple as running **make**.

populate_sysroot (stage)

The *populate_sysroot* task (stage is an alternate, easier to type name, that can be used to refer to this task) is responsible for making available libraries and headers (if any) that may be required by other packages to build. For example if you compile zlib then it's headers and the library need to be made available for other applications to include and link against.

> **Note:** This is different from the *install* task in that this is responsible for making available libraries and headers for use during build on the development host. Therefore it is libraries which normally have to stage things while applications normally don't need to. The *install* task on the other hand is making files available for packaging and ultimately installation on the target.

install

The *install* task is responsible for actually installing everything. This needs to install the software into the destination directory, **D**. This directory won't actually be a part of the final package though. In other words if you install something into **${D}/bin** then it will end up in the **/bin** directory in the package and therefore on the target.

package

The *package* task takes the installed files and splits them into separate directories under the **${WORKDIR}/install** directory, one per package. It moves the files for the destination directory, **${D}**, that they were installed in into the appropriate packages subdirectory. Usually there will be a main package, a separate documentation (-doc), development (-dev) and debugging packages (-dbg) for example.

package_write

The *package_write* task is responsible for taking each packages subdirectory and creating any actual installation package, such as .ipk, .deb or .rpm. Currently .ipk is the only fully supported packing format although .deb packages are being actively worked on. It should be reasonably easy for an experienced OpenEmbedded developer to add support for any other packaging formats they might required.

> **Note:** You'll notice that the bitbake output had tasks prefixed with *do_*, as in *do_install* vs *install*. This is slightly confusing but any task *x* is implemented via a function called *do_x* in the class or recipe where it is defined. Some places refer to the tasks via their name only and some with the *do* prefix.

You will almost certainly notice tasks beyond the ones above - there are various methods available to insert additional tasks into the tasks sequence. As an example the **insane.bbclass**, which performs various QA checks, does these checks by inserting a new task called *qa_configure* between the *configure* and *compile* tasks and another new task called *qa_staging* between *populate_sysroot* and *build* tasks. The former validates the result of the *configure* task and the later the results of the *populate_sysroot* task.

To determine the full list of tasks available for a specific recipe you can run bitbake on the recipe and asking it for the full list of available tasks:

```
$ bitbake -b recipes/perl/perl_5.8.8.bb -c listtasks
NOTE: package perl-5.8.8: started
NOTE: package perl-5.8.8-r11: task do_listtasks: started
do_fetchall
do_listtasks
do_rebuild
do_compile
do_build
do_populate_sysroot
do_mrproper
do_fetch
do_configure
do_clean
do_package
do_unpack
do_install
do_package_write
do_distribute_sources
do_showdata
do_qa_configure
do_qa_staging
do_patch
NOTE: package perl-5.8.8-r11: task do_listtasks: completed
NOTE: package perl-5.8.8: completed
$
```

If you're being observant you'll note that *listtasks* is in fact a task itself, and that the **-c** option to bitbake allows you to explicitly run specific tasks. We'll make use of this in the next section when we discuss working with a recipe.

7.5. Working with a single recipe

During development you're likely to often find yourself working on a single bitbake recipe - maybe trying to fix something or add a new version or perhaps working on a totally new recipe. Now that you know all about tasks you can use that knowledge to help speed up the development and debugging process.

Bitbake can be instructed to deal directly with a single recipe file by passing it via the **-b** parameter. This option takes the recipe as a parameter and instructs bitbake to process the named recipe only. Note that this ignores any dependencies that are in the recipe, so these must have already been built previously.

Here's a typical example that cleans up the package (using the *clean* task) and the rebuilds it with debugging output from bitbake enabled:

```
$ bitbake -b <bb-file> -c clean
$ bitbake -b <bb-file> -D
```

The options to bitbake that are most useful here are:

-b <bb-file>

The recipe to process;

-c <action>

The action to perform, typically the name of one of the tasks supported by the recipe;

-D

Display debugging information, use two **-D**'s for additional debugging;

-f

Force an operation. This is useful in getting bitbake to perform some operation it normally wouldn't do. For example, if you try and call the *compile* task twice in a row then bitbake will not do anything on the second attempt since it has already performed the task. By adding **-f** it will force it to perform the action regardless of if it thinks it's been done previously.

The most common actions (used with -c) are:

fetch

Try to download all of the required source files, but don't do anything else with them.

unpack

Unpack the source file but don't apply the patches yet. Sometimes you may want to look at the extracted, but not patched source code and that's what just unpacking will give you (sometimes handy to get diffs generated against the original source).

patch

Apply any patches.

configure

Performs any configuration that is required for the software.

compile

Perform the actual compilation steps of the software.

stage

If any files, such as header and libraries, will be required by other packages then they need to be installed into the staging area and that's what this task takes care of.

install

Install the software in preparation for packaging.

package

Package the software. Remember that this moves the files from the installation directory, D, into the packing install area. So to re-package you also need to re-install first.

clean

Delete the entire directory for this version of the software. Usually done to allow a test build with no chance of old files or changes being left behind.

Note that each of the actions that corresponds to a task will run any preceding tasks that have not yet been performed. So starting with compile will also perform the fetch, unpack, patch and configure actions.

A typical development session might involve editing files in the working directory and then recompiling until it all works:

```
[... test ...]
$ bitbake -b recipes/testapp/testapp_4.3.bb -c compile -D

[... save a copy of main.c and make some changes ...]
$ vi tmp/work/testapp-4.3-r0/main.c
$ bitbake -b recipes/testapp/testapp_4.3.bb -c compile -D -f

[... create a patch and add it to the recipe ...]
$ vi recipes/testapp/testapp_4.3.bb

[... test from clean ...]
$ bitbake -b recipes/testapp/testapp_4.3.bb -c clean
```

```
$ bitbake -b recipes/testapp/testapp_4.3.bb

[... NOTE: How to create the patch is not covered at this point ...]
```

Here's another example showing how you might go about fixing up the packaging in your recipe:

```
$ bitbake -b recipes/testapp/testapp_4.3.bb -c install -f
$ bitbake -b recipes/testapp/testapp_4.3.bb -c stage -f
$ find tmp/work/testapp_4.3/install
...
$ vi recipes/testapp/testapp_4.3.bb
```

At this stage you play with the **PACKAGE_** and **FILES_** variables and then repeat the above sequence.

Note how we install and then stage. This is one of those things where understanding the tasks helps a lot! Remember that stage moves the files from where they were installed into the various subdirectories (under **${WORKDIR}/install**) for each package. So if you try and run a stage task without a prior install there won't be any files there to stage! Note also that the stage tasks clears all the subdirectories in **${WORKDIR}/install** so you won't get any left over files. But beware, the install task doesn't clear **${D}** directory, so any left over files from a previous packing attempt will be left behind (which is ok if all you care about it staging).

7.6. Interactive bitbake

To interactively test things use:

```
$ bitbake -i
```

this will open the bitbake shell. From here there are a lot of commands available (try help).

First thing you will want to do is parse all of the recipes (recent bitbake version do this automatically when needed, so you don't need to manually do this anymore):

```
BB>> parse
```

You can now build a specific recipe:

```
BB>> build net-snmp
```

If it fails you may want to clean the build before trying again:

```
BB>> clean net-snmp
```

If you update the recipe by editing the .bb file (to fix some issues) then you will want to clean the package, reparse the modified recipe, and then build again:

```
BB>> clean net-snmp
BB>> reparse net-snmp
BB>> build net-snmp
```

Note that you can use wildcards in the bitbake shell as well:

```
BB>> build t*
```

7.7. Devshell

One of the areas in which OpenEmbedded helps you out is by setting various environment variables, such as **CC** and **PATH** etc, to values suitable for cross-compiling. If you wish to manually run configure scripts and compile files during development it would be nice to have all those values set for you. This is what devshell does - it provides you with an interactive shell with all the appropriate variables set for cross-compiling.

7.7.1. devshell via inherit

This is the newer method of obtaining a devshell and is the recommended way for most users now. The newer method requires that the devshell class be added to your configuration by inheriting it. This is usually done in your **local.conf** or your distributions conf file:

```
INHERIT += "src_distribute_local insane multimachine devshell"
```

With the inclusion of this class you'll find that devshell is added as a new task that you can use on recipes:

```
$ bitbake -b recipes/lzo/lzo_1.08.bb -c listtasks
NOTE: package lzo-1.08: started
NOTE: package lzo-1.08-r14: task do_listtasks: started
do_devshell
do_fetchall
do_listtasks
do_rebuild
do_compile
do_build
do_mrproper
do_fetch
do_configure
do_clean
do_populate_sysroot
```

```
do_package
do_unpack
do_install
do_package_write
do_distribute_sources
do_showdata
do_qa_staging
do_qa_configure
do_patch
NOTE: package lzo-1.08-r14: task do_listtasks: completed
NOTE: package lzo-1.08: completed
```

To bring up the devshell you call bitbake on a recipe and ask it for the devshell task:

```
$ bitbake -b recipes/lzo/lzo_1.08.bb -c devshell
NOTE: package lzo-1.08: started
NOTE: package lzo-1.08-r14: task do_devshell: started
[... devshell will appear here ...]
NOTE: package lzo-1.08-r14: task do_devshell: completed
NOTE: package lzo-1.08: completed
```

How the devshell appears depends on the settings of the **TERMCMD** variable - you can see the default settings and other possible values in **conf/bitbake.conf**. Feel free to try settings this to something else in your local.conf. Usually you will see a new terminal window open which is the devshell window.

The devshell task is inserted after the patch task, so if you have not already run bitbake on the recipe it will download the source and apply any patches prior to opening the shell.

> **Note:** This method of obtaining a devshell works if you using **bash** as your shell, it does not work if you are using **zsh** as your shell. Other shells may or may not work.

7.7.2. devshell addon

The devshell addon was the original method that was used to create a devshell.

It requires no changes to your configuration, instead you simply build the devshell recipe:

```
$ bitbake devshell
```

and then manually startup the shell. Once in the shell you'll usually want to change into the working directory for the recipe you are working on:

```
$ ./tmp/deploy/addons/sh4-linux-erouter-titan-devshell
bash: alias: `./configure': invalid alias name
[OE::sh4-linux-erouter-titan]:~$ cd tmp/work/lzo-1.08-r14/lzo-1.08
[OE::sh4-linux-erouter-titan]:~tmp/work/lzo-1.08-r14/lzo-1.08$
```

> **Note:** The name of the devshell addon depends on the target architecture, operating system and machine name. So you name will be different - just check for the appropriate name ending in -devshell.

7.7.3. Working in the devshell

[To be done]

7.8. Patching and patch management

[To be done]

Chapter 8. Recipes

8.1. Introduction

A bitbake recipe is a set of instructions that describes what needs to be done to retrieve the source code for some application, apply any necessary patches, provide any additional files (such as init scripts), compile it, install it and generate binary packages. The end result is a binary package that you can install on your target device, and maybe some intermediate files, such as libraries and headers, which can be used when building other applications.

In many ways the process is similar to creating .deb or .rpm packages for your standard desktop distributions with one major difference - in OpenEmbedded everything is being cross-compiled. This often makes the task far more difficult (depending on how well suited the application is to cross compiling), than it is for other packaging systems and sometimes impossible.

This chapter assumes that you are familiar with working with bitbake, including the work flow, required directory structures and bitbake configuration. If you are not familiar with these then first take a look at the chapter on bitbake usage.

8.2. Syntax of recipes

The basic items that make up a bitbake recipe file are:

functions

Functions provide a series of actions to be performed. Functions are usually used to override the default implementation of a task function, or to compliment (append or prepend to an existing function) a default function. Standard functions use sh shell syntax, although access to OpenEmbedded variables and internal methods are also available.

The following is an example function from the sed recipe:

```
do_install () {
    autotools_do_install
    install -d ${D}${base_bindir}
    mv ${D}${bindir}/sed ${D}${base_bindir}/sed.${PN}
}
```

It is also possible to implement new functions, that are not replacing or complimenting the default functions, which are called between existing tasks. It is also possible to implement functions in python instead of sh. Both of these options are not seen in the majority of recipes.

variable assignments and manipulations

Variable assignments allow a value to be assigned to a variable. The assignment may be static text or might include the contents of other variables. In addition to assignment, appending and prepending operations are also supported.

The following example shows some of the ways variables can be used in recipes:

```
S = "${WORKDIR}/postfix-${PV}"
PR = "r4"
CFLAGS += "-DNO_ASM"
SRC_URI_append = "file://fixup.patch"
```

keywords

Only a few keywords are used in bitbake recipes. They are used for things such as including common functions (*inherit*), loading parts of a recipe from other files (*include* and *require*) and exporting variables to the environment (export).

The following example shows the use of some of these keywords:

```
export POSTCONF = "${STAGING_BINDIR}/postconf"
inherit autoconf
require otherfile.inc
```

comments

Any lines that begin with a # are treated as comment lines and are ignored.

```
# This is a comment
```

The following is a summary of the most important (and most commonly used) parts of the recipe syntax:

Line continuation: \

To split a statement over multiple lines you should place a \ at the end of the line that is to be continued on the next line.

```
VAR = "A really long \
       line"
```

Note that there must not be anything (no spaces or tabs) after the \.

Using variables: ${...}

To access the contents of a variable you need to access it via *${<varname>}*:

```
SRC_URI = "${SOURCEFORGE_MIRROR}/libpng/zlib-${PV}.tar.gz"
```

Quote all assignments

All variable assignments should be quoted with double quotes. (It may work without them at present, but it will not work in the future).

```
VAR1 = "${OTHERVAR}"
VAR2 = "The version is ${PV}"
```

Conditional assignment

Conditional assignment is used to assign a value to a variable, but only when the variable is currently unset. This is commonly used to provide a default value for use when no specific definition is provided by the machine or distro configuration of the user's local.conf configuration.

The following example:

```
VAR1 ?= "New value"
```

will set **VAR1** to *"New value"* if it is currently empty. However if it was already set it would be unchanged. In the following **VAR1** is left with the value *"Original value"*:

```
VAR1 = "Original value"
VAR1 ?= "New value"
```

Appending: +=

You can append values to existing variables using the += operator. Note that this operator will add a space between the existing content of the variable and the new content.

```
SRC_URI += "file://fix-makefile.patch"
```

Prepending: =+

You can prepend values to existing variables using the =+ operator. Note that this operator will add a space between the new content and the existing content of the variable.

```
VAR =+ "Starts"
```

Appending: _append

You can append values to existing variables using the *_append* method. Note that this operator does not add any additional space, and it is applied after all the +=, and =+ operators have been applied.

The following example shows the space being explicitly added to the start to ensure the appended value is not merged with the existing value:

```
SRC_URI_append = " file://fix-makefile.patch"
```

The *_append* method can also be used with overrides, which results in the actions only being performed for the specified target or machine: [TODO: Link to section on overrides]

```
SRC_URI_append_sh4 = " file://fix-makefile.patch"
```

Note that the appended information is a variable itself, and therefore it's possible to use += or =+ to assign variables to the *_append* information:

```
SRC_URI_append = " file://fix-makefile.patch"
SRC_URI_append += "file://fix-install.patch"
```

Prepending: _prepend

You can prepend values to existing variables using the _prepend method. Note that this operator does not add any additional space, and it is applied after all the +=, and =+ operators have been applied.

The following example shows the space being explicitly added to the end to ensure the prepended value is not merged with the existing value:

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

The *_prepend* method can also be used with overrides, which result in the actions only being performed for the specified target or machine: [TODO: Link to section on overrides]

```
CFLAGS_prepend_sh4 = " file://fix-makefile.patch"
```

Note that the appended information is a variable itself, and therefore it's possible to use += or =+ to assign variables to the *_prepend* information:

```
CFLAGS_prepend = "-I${S}/myincludes "
CFLAGS_prepend += "-I${S}/myincludes2 "
```

Note also the lack of a space when using += to append to a prepend value - remember that the += operator adds space itself.

Spaces vs tabs

Spaces should be used for indentation, not hard tabs. Both currently work, however it is a policy decision of OE that spaces always be used.

Style: oe-stylize.py

To help with using the correct style in your recipes there is a python script in the contrib directory called *oe-stylize.py* which can be used to reformat your recipes to the correct style. The output will contain a list of warnings (to let you know what you did wrong) which should be edited out before using the new file.

```
$ contrib/oe-stylize.py myrecipe.bb > fixed-recipe.bb
vi fixed-recipe.bb
mv fixed.recipe.bb myrecipe.bb
```

Using python for complex operations: ${@...}

For more advanced processing it is possible to use python code during variable assignments, for doing search and replace on a variable for example.

Python code is indicated by a proceeding @ sign in the variable assignment.

```
CXXFLAGS := "${@'${CXXFLAGS}'.replace('-frename-registers', '')}"
```

More information about using python is available in the advanced python section.

Shell syntax

When describing a list of actions to take, shell syntax is used (as if you were writing a shell script). You should ensure that your script works with a generic sh and not require any bash (or other shell) specific functionality. The same applies to various system utilities (sed, grep, awk etc) that you may wish to use. If in doubt you should check with multiple implementations - including those from busybox.

For a detailed description of the syntax for the bitbake recipe files you should refer to the bitbake user manual.

8.3. Recipe naming: Names, versions and releases

Recipes in OpenEmbedded use a standard naming convention that includes the package name and version number in the filename. In addition to the name and version there is also a release number, which indicates changes to the way the package is built and/or packaged. The release number is contained within the recipe itself.

The expected format of recipe name is:

```
<package-name>_<version>.bb
```

where *<package-name>* is the name of the package (application, library, module, or whatever it is that is being packaged) and *version* is the version number.

So a typical recipe name would be:

```
strace_4.5.14.bb
```

which would be for version *4.5.14* of the *strace* application.

The release version is defined via the package release variable, PR, contained in the recipe. The expected format is:

```
r<n>
```

where *<n>* is an integer number starting from 0 initially and then incremented each time the recipe, or something that effects the recipe, is modified. So a typical definition of the release would be:

```
PR = "r1"
```

to specify release number *1* (the second release, the first would have been *0*). If there is no definition of PR in the recipe then the default value of "r0" is used.

Note: It is good practice to always define PR in your recipes, even for the *"r0"* release, so that when editing the recipe it is clear that the PR number needs to be updated.

You should always increment PR when modifying a recipe. Sometimes this can be avoided if the change will have no effect on the actual packages generated by the recipe, such as updating the SRC_URI to point to a new host. If in any doubt then you should increase the PR regardless of what has been changed.

The PR value should never be decremented. If you accidentally submit a large PR value for example then it should be left at the value and just increased for new releases, not reset back to a lower version.

When a recipe is being processed some variables are automatically set based on the recipe file name and can be used for other purposes from within the recipe itself. These include:

PN

The package name. Determined from the recipe filename - everything up until the first underscore is considered to be the package name. For the **strace_4.5.14.bb** recipe the PN variable would be set to *"strace"*.

PV

The package version. Determined from the recipe filename - everything between the first underscore and the final .bb is considered to be the package version. For the **strace_4.5.14.bb** recipe the PV variable would be set to *"4.5.14"*.

PR

The package release. This is explicitly set in the recipe. It defaults to "*r0"* if not set.

P

The package name and versions separated by a hyphen.

```
P = "${PN}-${PV}"
```

For the **strace_4.5.14.bb** recipe the P variable would be set to *"strace-4.5.14"*.

PF

The package name, version and release separated by hyphens.

```
PF = "${PN}-${PV}-${PR}"
```

For the **strace_4.5.14.bb recipe**, with PR set to *"r1"* in the recipe, the PF variable would be set to *"strace-4.5.14-r1"*.

While some of these variables are not commonly used in recipes (they are used internally though) both PN and PV are used a lot.

In the following example we are instructing the packaging system to include an additional directory in the package. We use PN to refer to the name of the package rather than spelling out the package name:

```
FILES_${PN} += "${sysconfdir}/myconf"
```

In the next example we are specifying the URL for the package source, by using PV in place of the actual version number it is possible to duplicate, or rename, the recipe for a new version without having to edit the URL:

```
SRC_URI = "ftp://ftp.vim.org/pub/vim/unix/vim-${PV}.tar.bz2"
```

8.4. Variables

One of the most confusing parts of bitbake recipes for new users is the large amount of variables that appear to be available to change and/or control the behaviour of some aspect of the recipe. Some variables, such as those derived from the file name are reasonably obvious, others are not at all obvious.

There are several places where these variables are derived from and/or used:

1. A large number of variables are defined in the bitbake configuration file conf/bitbake.conf - it's often a good idea to look through that file when trying to determine what a particular variable means.
2. Machine and distribution configuration files in conf/machine and conf/distro will sometimes define some variables specific to the machine and/or distribution. You should look at the appropriate files for your targets to see if anything is being defined that effects the recipes you are building.
3. Bitbake itself will define some variables. The FILE variable that defines the name of the bitbake recipe being processed is set by bitbake itself for example. Refer to the bitbake manual for more information on the variables that bitbake sets.
4. The classes that are used via the inherit keyword define and/or use the majority of the remaining variables. A class is like a library that contains parts of a bitbake recipe that is used by multiple recipes. To make them usable in more situations they often include a large number of variables to control how the class operates.

Another important aspect is that there are three different types of things that binaries and libraries are used for and they often have different variables for each. These include:

target

Refers to things built for the target and are expected to be run on the target device itself.

native

Refers to things built to run natively on the build host itself.

cross

Refers to things built to run natively on the build host itself, but produce output which is suitable for the target device. Cross versions of packages usually only exist for things like compilers and assemblers - i.e. things which are used to produce binary applications themselves.

8.5. Header

Practically all recipes start with a header section which describes various aspects of the package that is being built. This information is typically used directly by the package format (such as ipkg or deb) as meta data used to describe the package.

Variables used in the header include:

DESCRIPTION

Describes what the software does. Hopefully this gives enough information to a user to know if it's the right application for them.

The default description is: *"Version ${PV}-${PR} of package ${PN}"*.

HOMEPAGE

The URL of the home page of the application where new releases and more information can be found.

The default homepage is *"unknown"*.

SECTION

The section is used to categorize the application into a specific group. Often used by GUI based installers to help users when searching for software.

See SECTION variable for a list of the available sections.

The default section is *"base"*.

PRIORITY

The default priority is *"optional"*.

LICENSE

The license for the application. If it is not one of the standard licenses then the license itself must be included (where?).

As well as being used in the package meta-data the license is also used by the src_distribute class.

The default license is *"unknown"*.

8.6. Sources: Downloading, patching and additional files

A recipe's purpose is to describe how to take a software package and build it for your target device. The location of the source file (or files) is specified via the SRC_URI variable in the recipe. This can describe several types of URIs, the most common are:

http and https

Specifies files to be downloaded. A copy is stored locally so that future builds will not download the source again.

cvs, svn and git

Specifies that the files are to be retrieved using the specified version control system.

files

Plain files which are included locally. These can be used for adding documentation, init scripts or any other files that need to be added to build the package under openembedded.

patches

Plain files which are treated as patches and automatically applied.

If an http, https or file URI refers to a compressed file, an archive file or a compressed archive file, such as .tar.gz or .zip, then the files will be uncompressed and extracted from the archive automatically.

Archive files will be extracted from within the working directory, **${WORKDIR}** and plain files will be copied into the same directory. Patches will be applied from within the unpacked source directory, **${S}**. (Details on these directories is provided in the next section.)

The following example from the havp recipe shows a typical **SRC_URI** definition:

```
SRC_URI = "http://www.server-side.de/download/havp-${PV}.tar.gz \
           file://sysconfdir-is-etc.patch \
           file://havp.init \
           file://doc.configure.txt \
           file://volatiles.05_havp"
```

This describes several files

http://www.server-side.de/download/havp-${PV}.tar.gz

This is the URI of the havp source code. Note the use of the **${PV}** variable to specify the version. This is done to enable the recipe to be renamed for a new version without the need to edit the recipe itself. Because this is a .tar.gz compressed archive the file will be decompressed and extracted in the working dir **${WORKDIR}**.

file://sysconfdir-is-etc.patch

This is a local file that is used to patch the extracted source code. If a filename ends in .patch or .diff then this is treated as patch file and applied with striplevel 1. The patch will be applied from the unpacked source directory, **${S}**. In this case **${S}** will be **${WORKDIR}/havp-0.82**, and luckily the **havp-0.82.tar.gz** file extracts itself into that directory (so no need to explicitly change **${S}**).

file://havp.init file://doc.configure.txt file://volatiles.05_havp"

These are plain files which are just copied into the working directory **${WORKDIR}**. These are then used during the install task in the recipe to provide init scripts, documentation and volatiles configuration information for the package.

Full details on the **SRC_URI** variable and all the support URIs are available in the SRC_URI variable section of the reference chapter.

8.7. Directories: What goes where

A large part of the work of a recipe is involved with specifying where files are found and where they have to go. It's important for example that programs do not try and use files from **/usr/include** or **/usr/lib** since they are for the host system, not the target. Similarly you don't want programs installed into **/usr/bin** since that may overwrite your host system programs with versions that don't work on the host!

The following are some of the directories commonly referred to in recipes and will be described in more detail in the rest of this section:

Working directory: WORKDIR

This working directory for a recipe is where archive files will be extracted, plain files will be placed, subdirectories for logs, installed files etc will be created.

Unpacked source code directory: S

This is where patches are applied and where the program is expected to be compiled.

Destination directory: D

The destination directory. This is where your package should be installed into. The packaging system will then take the files from directories under here and package them up for installation on the target.

Installation directories: bindir, docdir, ...

There are a set of variables available to describe all of the paths on the target that you may want to use. Recipes should use these variables rather than hard coding any specific paths.

Staging directories: STAGING_LIBDIR, STAGING_INCDIR, ...

Staging directories are a special area for headers, libraries and other files that are generated by one recipe that may be needed by another recipe. A library package for example needs to make the library and headers available to other recipes so that they can link against them.

File path directories: FILE, FILE_DIRNAME, FILESDIR, FILESPATH

These directories are used to control where files are found. Understanding these can help you separate patches for different versions or releases of your recipes and/or use the same patch over multiple versions etc.

8.7.1. WORKDIR: The working directory

The working directory is where the source code is extracted, plain files (not patches) are copied and where the logs and installation files are created. A typical reason for needing to reference the work directory is for the handling of non patch files.

If we take a look at the recipe for quagga we can see example non patch files for configuration and init scripts:

```
SRC_URI = "http://www.quagga.net/download/quagga-${PV}.tar.gz \
           file://fix-for-lib-inpath.patch \
           file://quagga.init \
           file://quagga.default \
           file://watchquagga.init \
           file://watchquagga.default"
```

The recipe has two init files and two configuration files, which are not patches, but are actually files that it wants to include in the generated packages. Bitbake will copy these files into the working directory. So to access them during the install task we refer to them via the **WORKDIR** variable:

```
do_install () {
    # Install init script and default settings
    install -m 0755 -d ${D}${sysconfdir}/default ${D}${sysconfdir}/init.d ${D}${sysconfdir}
    install -m 0644 ${WORKDIR}/quagga.default ${D}${sysconfdir}/default/quagga
```

```
    install -m 0644 ${WORKDIR}/watchquagga.default ${D}${sysconfdir}/default/watchquagga
    install -m 0755 ${WORKDIR}/quagga.init ${D}${sysconfdir}/init.d/quagga
    install -m 0755 ${WORKDIR}/watchquagga.init ${D}${sysconfdir}/init.d/watchquagga
    ...
```

8.7.2. S: The unpacked source code directory

Bitbake expects to find the extracted source for a package in a directory called **<packagename>-<version>** in the **WORKDIR** directory. This is the directory it will change into before patching, compiling and installing the package.

For example, we have a package called **widgets_1.2.bb** which we are extracting from the **widgets-1.2.tar.gz** file. Bitbake expects the source to end up in a directory called **widgets-1.2** within the work directory. If the source does not end up in this directory then bitbake needs to be told this by explicitly setting **S**.

If **widgets-1.2.tar.gz** actually extracts into a directory called **widgets**, without the version number, instead of **widgets-1.2** then the **S** variable will be wrong and patching and/or compiling will fail. Therefore we need to override the default value of **S** to specify the directory the source was actually extracted into:

```
SRC_URI = "http://www.example.com/software/widgets-${PN}.tar.gz"
S = "${WORKDIR}/widgets"
```

8.7.3. D: The destination directory

The destination directory is where the completed application and all of its files are installed into in preparation for packaging. Typically an installation would place files in directories such as **/etc** and **/usr/bin** by default. Since those directories are used by the host system we do not want the packages to install into those locations. Instead they need to install into the directories below the destination directory.

So instead of installing into **/usr/bin** the package needs to install into **${D}/usr/bin**.

The following example from arpwatch shows the make install command being passed a **${D}** as the **DESTDIR** variable to control where the makefile installs everything:

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

The following example from quagga shows the use of the destination directory to install the configuration files and init scripts for the package:

```
do_install () {
        # Install init script and default settings
        install -m 0755 -d ${D}${sysconfdir}/default ${D}${sysconfdir}/init.d ${D}${sysconf
        install -m 0644 ${WORKDIR}/quagga.default ${D}${sysconfdir}/default/quagga
        install -m 0755 ${WORKDIR}/quagga.init ${D}${sysconfdir}/init.d/quagga
```

Note: You should not use directories such as **/etc** and **/usr/bin** directly in your recipes. You should use the variables that define these locations. The full list of these variables can be found in the Installation directories section of the reference chapter.

8.7.4. Staging directories

Staging is used to make libraries, headers and binaries available from the build of one recipe for use by another recipe. Building a library for example requires that packages be created containing the libraries and headers for development on the target as well as making them available on the host for building other packages that need the libraries and headers.

Making the libraries, headers and binaries available for use by other recipes on the host is called staging and is performed automatically derived from task *install*. Any recipes that contain items that are required additionally to build other packages should have a *install_append* task to make sure the items are all correctly placed into the staging area. The following example from clamav shows the clamav library and header being placed into the staging area:

```
do_install_append() {
        install -m 0755 -d ${D}${sysconfdir}/default/volatiles \
                           ${D}${sysconfdir}/init.d ${D}${docdir}/clamav

        # Save the installed clamd.conf in the doc dir and then install our new one
        install -m 0755 ${D}${sysconfdir}/clamd.conf ${D}${docdir}/clamav/clamd.conf.exampl
        install -m 0755 ${WORKDIR}/clamd.conf ${D}${sysconfdir}/clamd.conf

        # Save the installed freshclam.conf in the doc dir and then install our new one
        install -m 0755 ${D}${sysconfdir}/freshclam.conf ${D}${docdir}/clamav/freshclam.con

        # Install our config files and init scripts
        install -m 0755 ${WORKDIR}/freshclam.conf ${D}${sysconfdir}/freshclam.conf
        install -m 0755 ${WORKDIR}/clamav-daemon.init ${D}${sysconfdir}/init.d/clamav-daemo
        install -m 0755 ${WORKDIR}/clamav-freshclam.init ${D}${sysconfdir}/init.d/clamav-fr
```

```
        # We need some /var directories
        for i in 03_clamav-daemon 03_clamav-freshclam 03_clamav-data; do
          install -m 0644 ${WORKDIR}/volatiles.$i ${D}${sysconfdir}/default/volatiles/$i
        done
}
```

The following from the p3scan recipe shows the path to the clamav library and header being passed to the configure script. Without this the configure script would either fail to find the library, or worse still search the host system's directories for the library. Passing in the location results in it searching the correct location and finding the clamav library and headers:

```
EXTRA_OECONF = "--with-clamav=${STAGING_LIBDIR}/.. \
                --with-openssl=${STAGING_LIBDIR}/.. \
                --disable-ripmime"
```

While the staging directories are automatically added by OpenEmbedded to the compiler and linking commands it is sometimes necessary, as in the p3scan example above, to explicitly specify the location of the staging directories. Typically this is needed for autoconf scripts that search in multiple places for the libraries and headers.

> **Note:** Many of the helper classes, such as pkgconfig and autotools add appropriate commands to the stage task for you. Check with the individual class descriptions in the reference section to determine what each class is staging automatically for you.

A full list of staging directories can be found in the Staging directories section in the reference chapter.

8.7.5. FILESPATH/FILESDIR: Finding local files

The file related variables are used by bitbake to determine where to look for patches and local files.

Typically you will not need to modify these, but it is useful to be aware of the default values. In particular when searching for patches and/or files (file:// URIs), the default search path is:

${FILE_DIRNAME}/${PF}

> This is the package name, version and release, such as "**strace-4.5.14-r1**". This is very rarely used since the patches would only be found for the one exact release of the recipe.

${FILE_DIRNAME}/${P}

> This is the package name and version, such as "**strace-4.5.14**". This is by far the most common place to place version specific patches.

${FILE_DIRNAME}/${PN}

This is the package name only, such as "**strace**". This is not commonly used.

${FILE_DIRNAME}/files

This is just the directory called "**files**". This is commonly used for patches and files that apply to all versions of the package.

${FILE_DIRNAME}/

This is just the base directory of the recipe. This is very rarely used since it would just clutter the main directory.

Each of the paths is relative to **${FILE_DIRNAME}** which is the directory in which the recipe that is being processed is located.

The full set of variables that control the file locations are:

FILE

The path to the .bb file which is currently being processed.

FILE_DIRNAME

The path to the directory which contains the FILE which is currently being processed.

```
FILE_DIRNAME = "${@os.path.dirname(bb.data.getVar('FILE', d))}"
```

FILESPATH

The default set of directories which are available to use for the file:// URIs. Each directory is searched, in the specified order, in an attempt to find the file specified by each file:// URI:

```
FILESPATH = "${FILE_DIRNAME}/${PF}:${FILE_DIRNAME}/${P}:\
${FILE_DIRNAME}/${PN}:${FILE_DIRNAME}/files:${FILE_DIRNAME}"
```

FILESDIR

The default directory to search for file:// URIs. Only used if the file is not found in FILESPATH. This can be used to easily add one additional directory to the search path without having to modify the default FILESPATH setting. By default this is just the first directory from FILESPATH.

```
FILESDIR = "${@bb.which(bb.data.getVar('FILESPATH', d, 1), '.')}"
```

Sometimes recipes will modify the **FILESPATH** or **FILESDIR** variables to change the default search path for patches and files. The most common situation in which this is done is when one recipe includes another one in which the default values will be based on the name of the package doing the including, not the included package. Typically the included package will expect the files to be located in a directory based on its own name.

As an example the m4-native recipe includes the m4 recipe. This is fine, except that the m4 recipe expects its files and patches to be located in a directory called **m4**, while the native file name results in

them being searched for in **m4-native**. So the m4-native recipe sets the **FILESDIR** variable to the value of the actual m4 directory (where m4 itself has its files stored):

```
include m4_${PV}.bb
inherit native
FILESDIR = "${@os.path.dirname(bb.data.getVar('FILE',d,1))}/m4"
```

8.8. Basic examples

By now you should know enough about the bitbake recipes to be able to create a basic recipe. We'll cover a simple single file recipe and then a more advanced example that uses the autotools helper class (to be described later) to build an autoconf based package.

8.8.1. Hello world

Now it's time for our first recipe. This is going to be one of the simplest possible recipes: all code is included and there's only one file to compile and one readme file. While this isn't all that common, it's a useful example because it doesn't depend on any of the helper classes which can sometime hide a lot of what is going on.

First we'll create the myhelloworld.c file and a readme file. We'll place this in the files subdirectory, which is one of the places that is searched for file:// URIs:

```
$ mkdir recipes/myhelloworld
$ mkdir recipes/myhelloworld/files
$ cat > recipes/myhelloworld/files/myhelloworld.c
#include <stdio.h>

int main(int argc, char** argv)
{
        printf("Hello world!\n");
        return 0;
}
^D
$ cat > recipes/myhelloworld/files/README.txt
Readme file for myhelloworld.
^D
```

Now we have a directory for our recipe, recipes/myhelloworld, and we've created a files subdirectory in there to store our local files. We've created two local files, the C source code for our helloworld program and a readme file. Now we need to create the bitbake recipe.

First we need the header section, which will contain a description of the package and the release number. We'll leave the other header variables out for now:

```
DESCRIPTION = "My hello world program"
PR = "r0"
```

Next we need to tell it which files we want to be included in the recipe, which we do via file:// URIs and the SRC_URI variable:

```
SRC_URI = "file://myhelloworld.c \
           file://README.txt"
```

Note the use of the \ to continue a file and the use of file:// local URIs, rather than other types such as http://.

Now we need to provide a compile task which tells bitbake how to compile this program. We do this by defining a do_compile function in the recipe and providing the appropriate commands:

```
do_compile() {
        ${CC} ${CFLAGS} ${LDFLAGS} ${WORKDIR}/myhelloworld.c -o myhelloworld
}
```

Note the:

- use of the pre-defined compiler variables, **${CC}**, **${CFLAGS}** and **${LDFLAGS}**. These are set up automatically to contain the settings required to cross-compile the program for the target.
- use of **${WORKDIR}** to find the source file. As mentioned previously all files are copied into the working directory and can be referenced via the **${WORKDIR}** variable.

And finally we want to install the program and readme file into the destination directory so that it'll be packaged up correctly. This is done via the install task, so we need to define a do_install function in the recipe to describe how to install the package:

```
do_install() {
        install -m 0755 -d ${D}${bindir} ${D}${docdir}/myhelloworld
        install -m 0755 ${S}/myhelloworld ${D}${bindir}
        install -m 0644 ${WORKDIR}/README.txt ${D}${docdir}/myhelloworld
}
```

Note the:

- use of the **install** command to create directories and install the files, not cp.
- way directories are created before we attempt to install any files into them. The install command takes care of any subdirectories that are missing, so we only need to create the full path to the directory - no need to create the subdirectories.
- way we install everything into the destination directory via the use of the **${D}** variable.
- way we use variables to refer to the target directories, such as **${bindir}** and **${docdir}**.
- use of **${WORKDIR}** to get access to the **README.txt** file, which was provided via a file:// URI.

We'll consider this release 0 and version 0.1 of a program called helloworld. So we'll name the recipe myhelloworld_0.1.bb:

```
$ cat > recipes/myhelloworld/myhelloworld_0.1.bb
DESCRIPTION = "Hello world program"
PR = "r0"

SRC_URI = "file://myhelloworld.c \
           file://README.txt"

do_compile() {
        ${CC} ${CFLAGS} ${LDFLAGS} ${WORKDIR}/myhelloworld.c -o myhelloworld
}

do_install() {
        install -m 0755 -d ${D}${bindir} ${D}${docdir}/myhelloworld
        install -m 0644 ${S}/myhelloworld ${D}${bindir}
        install -m 0644 ${WORKDIR}/README.txt ${D}${docdir}/myhelloworld
}
^D
```

Now we are ready to build our package, hopefully it'll all work since it's such a simple example:

```
$ bitbake -b recipes/myhelloworld/myhelloworld_0.1.bb
NOTE: package myhelloworld-0.1: started
NOTE: package myhelloworld-0.1-r0: task do_fetch: started
NOTE: package myhelloworld-0.1-r0: task do_fetch: completed
NOTE: package myhelloworld-0.1-r0: task do_unpack: started
NOTE: Unpacking /home/lenehan/devel/oe/local-recipes/myhelloworld/files/helloworld.c to /ho
NOTE: Unpacking /home/lenehan/devel/oe/local-recipes/myhelloworld/files/README.txt to /home
NOTE: package myhelloworld-0.1-r0: task do_unpack: completed
NOTE: package myhelloworld-0.1-r0: task do_patch: started
NOTE: package myhelloworld-0.1-r0: task do_patch: completed
NOTE: package myhelloworld-0.1-r0: task do_configure: started
NOTE: package myhelloworld-0.1-r0: task do_configure: completed
NOTE: package myhelloworld-0.1-r0: task do_compile: started
NOTE: package myhelloworld-0.1-r0: task do_compile: completed
NOTE: package myhelloworld-0.1-r0: task do_install: started
NOTE: package myhelloworld-0.1-r0: task do_install: completed
NOTE: package myhelloworld-0.1-r0: task do_package: started
```

```
NOTE: package myhelloworld-0.1-r0: task do_package: completed
NOTE: package myhelloworld-0.1-r0: task do_package_write: started
NOTE: Not creating empty archive for myhelloworld-dbg-0.1-r0
Packaged contents of myhelloworld into /home/lenehan/devel/oe/build/titan-glibc-25/tmp/depl
Packaged contents of myhelloworld-doc into /home/lenehan/devel/oe/build/titan-glibc-25/tmp/
NOTE: Not creating empty archive for myhelloworld-dev-0.1-r0
NOTE: Not creating empty archive for myhelloworld-locale-0.1-r0
NOTE: package myhelloworld-0.1-r0: task do_package_write: completed
NOTE: package myhelloworld-0.1-r0: task do_populate_sysroot: started
NOTE: package myhelloworld-0.1-r0: task do_populate_sysroot: completed
NOTE: package myhelloworld-0.1-r0: task do_build: started
NOTE: package myhelloworld-0.1-r0: task do_build: completed
NOTE: package myhelloworld-0.1: completed
Build statistics:
  Attempted builds: 1
$
```

The package was successfully built, the output consists of two .ipkg files, which are ready to be installed on the target. One contains the binary and the other contains the readme file:

```
$ ls -l tmp/deploy/ipk/*/myhelloworld*
-rw-r--r--  1 lenehan lenehan 3040 Jan 12 14:46 tmp/deploy/ipk/sh4/myhelloworld_0.1-r0_sh4.
-rw-r--r--  1 lenehan lenehan  768 Jan 12 14:46 tmp/deploy/ipk/sh4/myhelloworld-doc_0.1-r0_
$
```

It's worthwhile looking at the working directory to see where various files ended up:

```
$ find tmp/work/myhelloworld-0.1-r0
tmp/work/myhelloworld-0.1-r0
tmp/work/myhelloworld-0.1-r0/myhelloworld-0.1
tmp/work/myhelloworld-0.1-r0/myhelloworld-0.1/patches
tmp/work/myhelloworld-0.1-r0/myhelloworld-0.1/myhelloworld
tmp/work/myhelloworld-0.1-r0/temp
tmp/work/myhelloworld-0.1-r0/temp/run.do_configure.21840
tmp/work/myhelloworld-0.1-r0/temp/log.do_stage.21840
tmp/work/myhelloworld-0.1-r0/temp/log.do_install.21840
tmp/work/myhelloworld-0.1-r0/temp/log.do_compile.21840
tmp/work/myhelloworld-0.1-r0/temp/run.do_stage.21840
tmp/work/myhelloworld-0.1-r0/temp/log.do_configure.21840
tmp/work/myhelloworld-0.1-r0/temp/run.do_install.21840
tmp/work/myhelloworld-0.1-r0/temp/run.do_compile.21840
tmp/work/myhelloworld-0.1-r0/install
tmp/work/myhelloworld-0.1-r0/install/myhelloworld-locale
tmp/work/myhelloworld-0.1-r0/install/myhelloworld-dbg
tmp/work/myhelloworld-0.1-r0/install/myhelloworld-dev
tmp/work/myhelloworld-0.1-r0/install/myhelloworld-doc
tmp/work/myhelloworld-0.1-r0/install/myhelloworld-doc/usr
tmp/work/myhelloworld-0.1-r0/install/myhelloworld-doc/usr/share
```

```
tmp/work/myhelloworld-0.1-r0/install/myhelloworld-doc/usr/share/doc
tmp/work/myhelloworld-0.1-r0/install/myhelloworld-doc/usr/share/doc/myhelloworld
tmp/work/myhelloworld-0.1-r0/install/myhelloworld-doc/usr/share/doc/myhelloworld/README.txt
tmp/work/myhelloworld-0.1-r0/install/myhelloworld
tmp/work/myhelloworld-0.1-r0/install/myhelloworld/usr
tmp/work/myhelloworld-0.1-r0/install/myhelloworld/usr/bin
tmp/work/myhelloworld-0.1-r0/install/myhelloworld/usr/bin/myhelloworld
tmp/work/myhelloworld-0.1-r0/image
tmp/work/myhelloworld-0.1-r0/image/usr
tmp/work/myhelloworld-0.1-r0/image/usr/bin
tmp/work/myhelloworld-0.1-r0/image/usr/share
tmp/work/myhelloworld-0.1-r0/image/usr/share/doc
tmp/work/myhelloworld-0.1-r0/image/usr/share/doc/myhelloworld
tmp/work/myhelloworld-0.1-r0/myhelloworld.c
tmp/work/myhelloworld-0.1-r0/README.txt
$
```

Things to note here are:

- The two source files are in **tmp/work/myhelloworld-0.1-r0**, which is the working directory as specified via the **${WORKDIR}** variable;
- There's logs of the various tasks in **tmp/work/myhelloworld-0.1-r0/temp** which you can look at for more details on what was done in each task;
- There's an image directory at **tmp/work/myhelloworld-0.1-r0/image** which contains just the directories that were to be packaged up. This is actually the destination directory, as specified via the **${D}** variable. The two files that we installed were originally in here, but during packaging they were moved into the install area into a subdirectory specific to the package that was being created (remember we have a main package and a -doc package being created.
- The program was actually compiled in the **tmp/work/myhelloworld-0.1-r0/myhelloworld-0.1** directory, this is the source directory as specified via the **${S}** variable.
- There's an install directory at **tmp/work/myhelloworld-0.1-r0/install** which contains the packages that were being generated and the files that go in the package. So we can see that the myhelloworld-doc package contains the single file **/usr/share/doc/myhelloworld/README.txt**, the myhelloworld package contains the single file **/usr/bin/myhelloworld** and the -dev, -dbg and -local packages are all empty.

At this stage it's good to verify that we really did produce a binary for the target and not for our host system. We can check that with the file command:

```
$ file tmp/work/myhelloworld-0.1-r0/install/myhelloworld/usr/bin/myhelloworld
tmp/work/myhelloworld-0.1-r0/install/myhelloworld/usr/bin/myhelloworld: ELF 32-bit LSB exec
$ file /bin/ls
/bin/ls: ELF 64-bit LSB executable, AMD x86-64, version 1 (SYSV), for GNU/Linux 2.4.0, dyna
$
```

This shows us that the helloworld program is for an SH processor (obviously this will change depending on what your target system is), while checking the **/bin/ls** program on the host shows us that the host system is an AMD X86-64 system. That's exactly what we wanted.

8.8.2. An autotools package

Now for an example of a package that uses autotools. These are programs that you need to run a configure script for, passing various parameters, and then make. To make these work when cross-compiling you need to provides a lot of variables to the configure script. But all the hard work has already been done for you. There's an autotools class which takes care of most of the complexity of building an autotools based package.

Let's take a look at the tuxnes recipe which is an example of a very simple autotools based recipe:

```
$ cat recipes/tuxnes/tuxnes_0.75.bb
DESCRIPTION = "Tuxnes Nintendo (8bit) Emulator"
HOMEPAGE = "http://prdownloads.sourceforge.net/tuxnes/tuxnes-0.75.tar.gz"
LICENSE = "GPLv2"
SECTION = "x/games"
PRIORITY = "optional"
PR = "r1"

SRC_URI = "http://heanet.dl.sourceforge.net/sourceforge/tuxnes/tuxnes-0.75.tar.gz"

inherit autotools
```

This is a really simple recipe. There's the standard header that describes the package. Then the SRC_URI, which in this case is an http URL that causes the source code to be downloaded from the specified URI. And finally there's an "**inherit autotools**" command which loads the autotools class. The autotools class will take care of generating the required configure, compile and install tasks. So in this case there's nothing else to do - that's all there is to it.

It would be nice if it was always this simple. Unfortunately there's usually a lot more involved for various reasons including the need to:

- Pass parameters to configure to enable and disable features;
- Pass parameters to configure to specify where to find libraries and headers;
- Make modifications to prevent searching for headers and libraries in the normal locations (since they belong to the host system, not the target);
- Make modifications to prevent the configure script from trying to compile and run programs - any programs it compiles will be for the target and not the host and so cannot be run.
- Manually implement staging scripts;
- Deal with lots of other more complex issues;

Some of these items are covered in more detail in the advanced autoconf section.

8.9. Dependencies: What's needed to build and/or run the package?

Dependencies should be familiar to anyone who has used an .rpm and .deb based desktop distribution. A dependency is something that a package requires either to run the package (a run-time dependency) or to build the package (a build-time or compile-time, dependency).

There are two variables provided to allow the specifications of dependencies:

DEPENDS

Specifies build-time dependencies, via a list of bitbake recipes to build prior to building the recipe. These are programs (flex-native) or libraries (libpcre) that are required in order to build the package.

RDEPENDS

Specifies run-time dependencies, via a list of packages to install prior to installing the current package. These are programs or libraries that are required in order to run the program. Note that libraries which are dynamically linked to an application will be automatically detected and added to **RDEPENDS** and therefore do not need to be explicitly declared. If a library was dynamically loaded then it would need to be explicitly listed.

If we take openssh for an example, it requires zlib and openssl in order to both build and run. In the recipe we have:

```
DEPENDS = "zlib openssl"
```

This tells bitbake that it will need to build and stage zlib and openssl prior to trying to build openssh, since openssh requires both of them. Note that there is no **RDEPENDS** even though openssh requires both of them to run. The run time dependencies on libz1 (the name of the package containing the zlib library) and libssl0 (the name of the package containing the ssl library) are automatically determined and added via the auto shared libs dependency code.

8.10. Methods: Built-in methods to make your life easier

There are several helper functions defined by the base class, which is included by default for all recipes. Many of these are used a lot in both recipes and other classes.

The most commonly seen, and most useful functions, include:

oe_runmake

This function is used to run make. However unlike calling make yourself this will pass the EXTRA_OEMAKE settings to make, will display a note about the make command and will check for any errors generated via the call to make.

You should never have any reason to call make directly and should also use oe_runmake when you need to run make.

oe_runconf (autotools only)

This function is used to run the configure script of a package that is using the autotools class. This takes care of passing all of the correct parameters for cross-compiling and for installing into the appropriate target directory.

It also passes the value of the **EXTRA_OECONF** variable to the configure script. For many situations setting **EXTRA_OECONF** is sufficient and you'll have no need to define your own configure task in which you call oe_runconf manually.

If you need to write your own *configure* task for an autotools package you can use oe_runconf to manually call the configure process when it is required. The following example from net-snmp shows oe_runconf being called manually so that the parameter for specifying the endianess can be computed and passed in to the configure script:

```
do_configure() {
    # Additional flag based on target endianess (see siteinfo.bbclass)
    ENDIANESS="${@base_conditional('SITEINFO_ENDIANNESS', 'le', '--with-endianness=litt
    oenote Determined endianess as: $ENDIANESS
    oe_runconf $ENDIANESS
}
```

oe_libinstall

This function is used to install **.so**, **.a** and associated libtool **.la** libraries. It will determine the appropriate libraries to install and take care of any modifications that may be required for **.la** files.

This function supports the following options:

-C <dir>

Change into the specified directory before attempting to install a library. Used when the libraries are in subdirectories of the main package.

-s

Require the presence of a **.so** library as one of the libraries that is installed.

-a

Require the presence of a **.a** library as one of the libraries that is installed.

The following example from gdbm shows the installation of **.so**, **.a** (and associated **.la**) libraries into the staging library area:

```
do_stage () {
```

```
    oe_libinstall -so -a libgdbm ${STAGING_LIBDIR}
    install -m 0644 ${S}/gdbm.h ${STAGING_INCDIR}/
}
```

oenote

Used to display informational messages to the user.

The following example from net-snmp uses oenote to tell the user which endianess it determined was appropriate for the target device:

```
do_configure() {
    # Additional flag based on target endianess (see siteinfo.bbclass)
    ENDIANESS="${@base_conditional('SITEINFO_ENDIANNESS', 'le', '--with-endianness=litt
    oenote Determined endianess as: $ENDIANESS
    oe_runconf $ENDIANESS
}
```

oewarn

Used to display a warning message to the user, warning of something that may be problematic or unexpected.

oedebug

Used to display debugging related information. These messages will only be visible when bitbake is run with the **-D** flag to enable debug output.

oefatal

Used to display a fatal error message to the user, and then abort the bitbake run.

The following example from linux-libc-headers shows the use of oefatal to tell the user when it cannot find the kernel source code for the specified target architecture:

```
do_configure () {
    case ${TARGET_ARCH} in
        alpha*)   ARCH=alpha ;;
        arm*)     ARCH=arm ;;
        cris*)    ARCH=cris ;;
        hppa*)    ARCH=parisc ;;
        i*86*)    ARCH=i386 ;;
        ia64*)    ARCH=ia64 ;;
        mips*)    ARCH=mips ;;
        m68k*)    ARCH=m68k ;;
        powerpc*) ARCH=ppc ;;
        s390*)    ARCH=s390 ;;
        sh*)      ARCH=sh ;;
        sparc64*) ARCH=sparc64 ;;
        sparc*)   ARCH=sparc ;;
        x86_64*)  ARCH=x86_64 ;;
    esac
```

```
    if test !  -e include/asm-$ARCH; then
        oefatal unable to create asm symlink in kernel headers
    fi
...
```

base_conditional (python)

The base conditional python function is used to set a variable to one of two values based on the definition of a third variable. The general usage is:

```
${@base_conditional('<variable-name>', '<value>', '<true-result>', '<false-result>', d)
```

where:

variable-name

This is the name of a variable to check.

value

This is the value to compare the variable against.

true-result

If the variable equals the value then this is what is returned by the function.

false-result

If the variable does not equal the value then this is what is returned by the function.

Note: The ${@...} syntax is used to call python functions from within a recipe or class. This is described in more detail in the advanced python section.

The following example from the openssl recipe shows the addition of either **-DL_ENDIAN** or **-DB_ENDIAN** depending on the value of **SITEINFO_ENDIANNESS** which is set to le for little endian targets and to be for big endian targets:

```
do_compile () {
    ...
    # Additional flag based on target endianess (see siteinfo.bbclass)
    CFLAG="${CFLAG} ${@base_conditional('SITEINFO_ENDIANNESS', 'le', '-DL_ENDIAN', '-DB_
    ...
```

8.11. Packaging: Defining packages and their contents

A bitbake recipe is a set of instructions for creating one, or more, packages for installation on the target device. Typically these are .ipkg or .deb packages (although bitbake itself isn't associated with any particular packaging format).

By default several packages are produced automatically without any special action required on the part of the recipe author. The following example of the packaging output from the helloworld example above shows this packaging in action:

```
[NOTE: package helloworld-0.1-r0: task do_package_write: started
NOTE: Not creating empty archive for helloworld-dbg-0.1-r0
Packaged contents of helloworld into /home/lenehan/devel/oe/build/titan-glibc-25/tmp/deploy
Packaged contents of helloworld-doc into /home/lenehan/devel/oe/build/titan-glibc-25/tmp/de
NOTE: Not creating empty archive for helloworld-dev-0.1-r0
NOTE: Not creating empty archive for helloworld-locale-0.1-r0
NOTE: package helloworld-0.1-r0: task do_package_write: completed
```

We can see from above that the packaging did the following:

- Created a main package, **helloworld_0.1-r0_sh4.ipk**. This package contains the helloworld binary **/usr/bin/helloworld**.
- Created a documentation package, **helloworld-doc_0.1-r0_sh4.ipk**. This package contains the readme file **/usr/share/doc/helloworld/README.txt**.
- Considered creating a debug package, **helloworld-dbg-0.1-r0_sh4.ipk**, a development package **helloworld-dev-0.1-r0_sh4.ipk** and a locale package **helloworld-locale-0.1-r0_sh4.ipk**. It didn't create the packages due to the fact that it couldn't find any files that would actually go in the packages.

There are several things happening here which are important to understand:

1. There is a default set of packages that are considered for creation. This set of packages is controlled via the **PACKAGES** variable.
2. For each package there is a default set of files and/or directories that are considered to belong to those packages. The documentation packages for example include anything found **/usr/share/doc**. The set of files and directories are controlled via the **FILES_<package-name>** variables.
3. By default packages that contain no files are not created and no error is generated. The decision to create empty packages or not is controlled via the **ALLOW_EMPTY** variable.

8.11.1. Philosophy

Separate packaging, where possible, is of high importance in OpenEmbedded. Many of the target devices have limited storage space and RAM and giving distributions and users the option of not installing a part of the package they don't need allows them to reduce the amount of storage space required.

As an example almost no distributions will include documentation or development libraries since they are not required for the day to day operation of the device. In particular if your package provides multiple binaries, and it would be common to only use one or the other, then you should consider separating them into separate packages.

By default several groups of files are automatically separated out, including:

dev

Any files required for development. This includes header files, static libraries, the shared library symlinks required only for linking etc. These would only ever need to be installed by someone attempting to compile applications on the target device. While this does happen it is very uncommon and so these files are automatically moved into a separate package

doc

Any documentation related files, including man pages. These are files which are of informational purposes only. For many embedded devices there is no way for the user to see any of the documentation anyway, and documentation can consume a lot of space. By separating these out they don't take any space by default but distributions and/or users may choose to install them if they need some documentation on a specific package.

locale

Locale information provides translation information for packages. Many users do not require these translations, and many devices will only want to provide them for user visible components, such as UI related items, and not for system binaries. By separating these out it is left up to the distribution or users to decide if they are required or not.

8.11.2. Default packages and files

The default package settings are defined in **conf/bitbake.conf** and are suitable for a lot of recipes without any changes. The following list shows the default values for the packaging related variables:

PACKAGES

This variable lists the names of each of the packages that are to be generated.

```
PACKAGES = "${PN}-dbg ${PN} ${PN}-doc ${PN}-dev ${PN}-locale"
```

Note that the order of packages is important: the packages are processed in the listed order. So if two packages specify the same file then the first package listed in packages will get the file. This is important when packages use wildcards to specify their contents.

For example if the main package, **${PN}**, contains **/usr/bin/*** (i.e. all files in **/usr/bin**), but you want **/usr/bin/tprogram** in a separate package, **${PN}-tpackage**, you would need to either ensure that **${PN}-tpackage** is listed prior to **${PN}** in **PACKAGES** or that **FILES_${PN}** was modified to not contain the wildcard that matches **/usr/bin/tprogram**.

Note that the -dbg package contains the debugging information that has been extracted from binaries and libraries prior to them being stripped. This package should always be the first package in the package list to ensure that the debugging information is correctly extracted and moved to the package prior to any other packaging decisions being made.

FILES_${PN}

The base package, this includes everything needed to actually run the application on the target system.

```
FILES_${PN} = "\
    ${bindir}/* \
    ${sbindir}/* \
    ${libexecdir}/* \
    ${libdir}/lib*.so.* \
    ${sysconfdir} \
    ${sharedstatedir} \
    ${localstatedir} \
    /bin/* \
    /sbin/* \
    /lib/*.so* \
    ${datadir}/${PN} \
    ${libdir}/${PN}/* \
    ${datadir}/pixmaps \
    ${datadir}/applications \
    ${datadir}/idl \
    ${datadir}/omf \
    ${datadir}/sounds \
    ${libdir}/bonobo/servers"
```

FILES_${PN}-dbg

The debugging information extracted from non-stripped versions of libraries and executable's. OpenEmbedded automatically extracts the debugging information into files in .debug directories and then strips the original files.

```
FILES_${PN}-dbg = "\
    ${bindir}/.debug \
    ${sbindir}/.debug \
    ${libexecdir}/.debug \
    ${libdir}/.debug \
    /bin/.debug \
    /sbin/.debug \
    /lib/.debug \
    ${libdir}/${PN}/.debug"
```

FILES_${PN}-doc

Documentation related files. All documentation is separated into its own package so that it does not need to be installed unless explicitly required.

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

FILES_${PN}-dev

Development related files. Any headers, libraries and support files needed for development work on the target.

```
FILES_${PN}-dev = "\
    ${includedir} \
    ${libdir}/lib*.so \
    ${libdir}/*.la \
    ${libdir}/*.a \
    ${libdir}/*.o \
    ${libdir}/pkgconfig \
    /lib/*.a \
    /lib/*.o \
    ${datadir}/aclocal"
```

FILES_${PN}-locale

Locale related files.

```
FILES_${PN}-locale = "${datadir}/locale"
```

8.11.3. Wildcards

Wildcards used in the **FILES** variables are processed via the python function **fnmatch**. The following items are of note about this function:

- **/<dir>/***: This will match all files and directories in the **dir** - it will not match other directories.
- **/<dir>/a***: This will only match files, and not directories.
- **/dir**: will include the directory **dir** in the package, which in turn will include all files in the directory and all subdirectories.

Note that the order of packages effects the files that will be matched via wildcards. Consider the case where we have three binaries in the **/usr/bin** directory and we want the test program in a separate package:

```
/usr/bin/programa /usr/bin/programb /usr/bin/test
```

So we define a new package and instruct bitbake to include /usr/bin/test in it.

```
FILES-${PN}-test = "${bindir}/test"
PACKAGES += "FILES-${PN}-test"
```

When the package is regenerated no **${PN}-test** package will be created. The reason for this is that the **PACKAGES** line now looks like this:

```
{PN}-dbg ${PN} ${PN}-doc ${PN}-dev ${PN}-locale ${PN}-test
```

Note how **${PN}** is listed prior to **${PN}-test**, and if we look at the definition of **FILES-${PN}** it contains the **${bindir}/*** wildcard. Since **${PN}** is first it'll match that wildcard and be moved into the **${PN}** package prior to processing of the **${PN}-test** package.

To achieve what we are trying to accomplish we have two options:

1. Modify the definition of **${PN}** so that the wildcard does not match the test program.

   We could do this for example:

   ```
   FILES-${PN} = "${bindir}/p*"
   ```

   So now this will only match things in the bindir that start with p, and therefore not match our test program. Note that **FILES-${PN}** contains a lot more entries and we'd need to add any of the others that refer to files that are to be included in the package. In this case we have no other files, so it's safe to do this simple declaration.

2. Modify the order of packages so that the **${PN}-test** package is listed first.

   The most obvious way to do this would be to prepend our new package name to the packages list instead of appending it:

   ```
   PACKAGES =+ "FILES-${PN}-test"
   ```

   In some cases this would work fine, however there is a problem with this for packages that include binaries. The package will now be listed before the -dbg package and often this will result in the .debug directories being included in the package. In this case we are explicitly listing only a single file (and not using wildcards) and therefore it would be ok.

   In general it's more common to have to redefine the entire package list to include your new package plus any of the default packages that you require:

   ```
   PACKAGES = "${PN}-dbg ${PN}-test ${PN} ${PN}-doc ${PN}-dev ${PN}-locale"
   ```

8.11.4. Checking the packages

During recipe development it's useful to be able to check on exactly what files went into each package, which files were not packaged and which packages contain no files.

One of the easiest methods is to run find on the install directory. In the install directory there is one subdirectory created per package, and the files are moved into the install directory as they are matched to a specific package. The following shows the packages and files for the helloworld example:

```
$ find tmp/work/helloworld-0.1-r0/install
tmp/work/helloworld-0.1-r0/install
tmp/work/helloworld-0.1-r0/install/helloworld-locale
tmp/work/helloworld-0.1-r0/install/helloworld-dbg
```

```
tmp/work/helloworld-0.1-r0/install/helloworld-dev
tmp/work/helloworld-0.1-r0/install/helloworld-doc
tmp/work/helloworld-0.1-r0/install/helloworld-doc/usr
tmp/work/helloworld-0.1-r0/install/helloworld-doc/usr/share
tmp/work/helloworld-0.1-r0/install/helloworld-doc/usr/share/doc
tmp/work/helloworld-0.1-r0/install/helloworld-doc/usr/share/doc/helloworld
tmp/work/helloworld-0.1-r0/install/helloworld-doc/usr/share/doc/helloworld/README.txt
tmp/work/helloworld-0.1-r0/install/helloworld
tmp/work/helloworld-0.1-r0/install/helloworld/usr
tmp/work/helloworld-0.1-r0/install/helloworld/usr/bin
tmp/work/helloworld-0.1-r0/install/helloworld/usr/bin/helloworld
$
```

The above shows that the -local, -dbg and -dev packages are all empty, and the -doc and base package contain a single file each. Using the "**-type f**" option to find to show just files will make this clearer as well.

In addition to the install directory the image directory (which corresponds to the destination directory, **D**) will contain any files that were not packaged:

```
$ find tmp/work/helloworld-0.1-r0/image
tmp/work/helloworld-0.1-r0/image
tmp/work/helloworld-0.1-r0/image/usr
tmp/work/helloworld-0.1-r0/image/usr/bin
tmp/work/helloworld-0.1-r0/image/usr/share
tmp/work/helloworld-0.1-r0/image/usr/share/doc
tmp/work/helloworld-0.1-r0/image/usr/share/doc/helloworld
$
```

In this case all files were packaged and so there are no left over files. Using find with "**-type f**" makes this much clearer:

```
$ find tmp/work/helloworld-0.1-r0/image -type f
$
```

Messages regarding missing files are also displayed by bitbake during the package task:

```
NOTE: package helloworld-0.1-r0: task do_package: started
NOTE: the following files were installed but not shipped in any package:
NOTE:   /usualdir/README.txt
NOTE: package helloworld-0.1-r0: task do_package: completed
```

Except in very unusual circumstances there should be no unpackaged files left behind by a recipe.

8.11.5. Excluding files

There's no actual support for explicitly excluding files from packaging. You could just leave them out of any package, but then you'll get warnings (or errors if requesting full package checking) during packaging which is not desirable. It also doesn't let other people know that you've deliberately avoided packaging the file or files.

In order to exclude a file totally you should avoid installing it in the first place during the install task.

In some cases it may be easier to let the package install the file and then explicitly remove the file at the end of the install task. The following example from the samba recipe shows the removal of several files that get installed via the default install task generated by the autotools class. By using *do_install_append* these commands are run after the autotools generated install task:

```
do_install_append() {
    ...
    rm -f ${D}${bindir}/*.old
    rm -f ${D}${sbindir}/*.old
    ...
}
```

8.11.6. Debian naming

A special *debian library name* policy can be applied for packages that contain a single shared library. When enabled packages will be renamed to match the debian policy for such packages.

Debian naming is enabled by including the debian class via either **local.conf** or your distribution's configuration file:

```
INHERIT += "debian"
```

The policy works by looking at the shared library name and version and will automatically rename the package to <*libname*><*lib-major-version*>. For example if the package name (PN) is **foo** and the package ships a file named **libfoo.so.1.2.3** then the package will be renamed to **libfoo1** to follow the debian policy.

If we look at the *lzo_1.08.bb* recipe, currently at release 14, it generates a package containing a single shared library :

```
$ find tmp/work/lzo-1.08-r14/install/
tmp/work/lzo-1.08-r14/install/lzo
tmp/work/lzo-1.08-r14/install/lzo/usr
```

```
tmp/work/lzo-1.08-r14/install/lzo/usr/lib
tmp/work/lzo-1.08-r14/install/lzo/usr/lib/liblzo.so.1
tmp/work/lzo-1.08-r14/install/lzo/usr/lib/liblzo.so.1.0.0
```

Without debian naming this package would have been called **lzo_1.08-r14_sh4.ipk** (and the corresponding dev and dbg packages would have been called **lzo-dbg_1.08-r14_sh4.ipk** and **lzo-dev_1.08-r14_sh4.ipk**). However with debian naming enabled the package is renamed based on the name of the shared library, which is **liblzo.so.1.0.0** in this case. So the name **lzo** is replaced with **liblzo1**:

```
$ find tmp/deploy/ipk/ -name '*lzo*'
tmp/deploy/ipk/sh4/liblzo1_1.08-r14_sh4.ipk
tmp/deploy/ipk/sh4/liblzo-dev_1.08-r14_sh4.ipk
tmp/deploy/ipk/sh4/liblzo-dbg_1.08-r14_sh4.ipk
```

Some variables are available which effect the operation of the debian renaming class:

LEAD_SONAME

If the package actually contains multiple shared libraries then one will be selected automatically and a warning will be generated. This variable is a regular expression which is used to select which shared library from those available is to be used for debian renaming.

DEBIAN_NOAUTONAME_<pkgname>

If this variable is set to 1 for a package then debian renaming will not be applied for the package.

AUTO_LIBNAME_PKGS

If set this variable specifies the prefix of packages which will be subject to debian renaming. This can be used to prevent all of the packages being renamed via the renaming policy.

8.11.7. Empty packages

By default empty packages are ignored. Occasionally you may wish to actually create empty packages, typically done when you want a virtual package which will install other packages via dependencies without actually installing anything itself. The **ALLOW_EMPTY** variable is used to control the creation of empty packages:

ALLOW_EMPTY

Controls if empty packages will be created or not. By default this is **"0"** and empty packages are not created. Setting this to **"1"** will permit the creation of empty packages (packages containing no files).

8.12. Tasks: Playing with tasks

Bitbake steps through a series of tasks when building a recipe. Sometimes you need to explicitly define what a class does, such as providing a **do_install** function to implement the *install* task in a recipe and sometimes they are provided for you by common classes, such as the autotools class providing the default implementations of *configure*, *compile* and *install* tasks.

There are several methods available to modify the tasks that are being run:

Overriding the default task implementation

By defining your own implementation of a task you'll override any default or class provided implementations.

For example, you can define your own implementation of the compile task to override any default implementation:

```
do_compile() {
      oe_runmake DESTDIR=${D}
}
```

If you wish to totally prevent the task from running you need to define your own empty implementation. This is typically done via the definition of the task using a single colon:

```
do_configure() {
    :
}
```

Appending or prepending to the task

Sometimes you want the default implementation, but you require additional functionality. This can done by appending or pre-pending additional functionality onto the task.

The following example from units shows an example of installing an addition file which for some reason was not installed via the autotools normal *install* task:

```
do_install_append() {
       install -d ${D}${datadir}
       install -m 0655 units.dat ${D}${datadir}
}
```

The following example from the cherokee recipe shows an example of adding functionality prior to the default *install* task. In this case it compiles a program that is used during installation natively so that it will work on the host. Without this the autotools default *install* task would fail since it'd try to run the program on the host which was compiled for the target:

```
do_install_prepend () {
        # It only needs this app during the install, so compile it natively
```

```
        $BUILD_CC -DHAVE_SYS_STAT_H -o cherokee_replace cherokee_replace.c
}
```

Defining a new task

Another option is to define a totally new task, and then register that with bitbake so that it runs in between two of the existing tasks.

The following example shows a situation in which a cvs tree needs to be copied over the top of an extracted tar.gz archive, and this needs to be done before any local patches are applied. So a new task is defined to perform this action, and then that task is registered to run between the existing *unpack* and *patch* tasks:

```
do_unpack_extra(){
    cp -pPR ${WORKDIR}/linux/* ${S}
}
addtask unpack_extra after do_unpack before do_patch
```

Note: The task to add does not have the do_ prepended to it, however the tasks to insert it after and before do have the do_ prepended. No errors will be generated if this is wrong, the additional task simply won't be executed.

Using overrides

Overrides (described fully elsewhere) allow for various functionality to be performed conditionally based on the target machine, distribution, architecture etc.

While not commonly used it is possible to use overrides when defining tasks. The following example from udev shows an additional file being installed for the specified machine only by performing an append to the *install* task for the h2200 machine only:

```
do_install_append_h2200() {
    install -m 0644 ${WORKDIR}/50-hostap_cs.rules         ${D}${sysconfdir}/udev/rules.d
}
```

8.13. Classes: The separation of common functionality

Often a certain pattern is followed in more than one recipe, or maybe some complex python based functionality is required to achieve the desired end result. This is achieved through the use of classes, which can be found in the classes subdirectory at the top-level of an OE checkout.

Being aware of the available classes and understanding their functionality is important because classes:

- Save developers time by performing actions that they would otherwise need to perform themselves;
- Perform a lot of actions in the background making a lot of recipes difficult to understand unless you are aware of classes and how they work;
- A lot of detail on how things work can be learnt from looking at how classes are implemented.

A class is used via the inherit method. The following is an example for the *curl* recipe showing that it uses three classes:

```
inherit autotools pkgconfig binconfig
```

In this case it is utilizing the services of three separate classes:

autotools

The autotools class is used by programs that use the GNU configuration tools and takes care of the configuration and compilation of the software;

pkgconfig

The pkgconfig class is used to stage the *.pc* files which are used by the **pkg-config** program to provide information about the package to other software that wants to link to this software;

binconfig

The binconfig class is used to stage the *<name>-config* files which are used to provide information about the package to other software that wants to link to this software;

Each class is implemented via the file in the **classes** subdirectory named **<classname>.bbclass** and these can be examined for further details on a particular class, although sometimes it's not easy to understand everything that's happening. Many of the classes are covered in detail in various sections in this user manual.

8.14. Staging: Making includes and libraries available for building

Staging is the process of making files, such as include files and libraries, available for use by other recipes. This is different to installing because installing is about making things available for packaging and then eventually for use on the target device. Staging on the other hand is about making things available on the host system for use in building later applications.

Taking bzip2 as an example you can see that it stages a header file and its library files:

```
do_stage () {
    install -m 0644 bzlib.h ${STAGING_INCDIR}/
    oe_libinstall -a -so libbz2 ${STAGING_LIBDIR}
}
```

The *oe_libinstall* method used in the bzip2 recipe is described in the methods section, and it takes care of installing libraries (into the staging area in this case). The staging variables are automatically defined to the correct staging location, in this case the main staging variables are used:

STAGING_INCDIR

The directory into which staged header files should be installed. This is the equivalent of the standard **/usr/include** directory.

STAGING_LIBDIR

The directory into which staged library files should be installed. This is the equivalent of the standard **/usr/lib** directory.

Additional staging related variables are covered in the Staging directories section in Chapter 9.

Looking in the staging area under tmp you can see the result of the bzip2 recipes staging task:

```
$ find tmp/staging -name '*bzlib*'
tmp/staging/sh4-linux/include/bzlib.h
$ find tmp/staging -name '*libbz*'
tmp/staging/sh4-linux/lib/libbz2.so
tmp/staging/sh4-linux/lib/libbz2.so.1.0
tmp/staging/sh4-linux/lib/libbz2.so.1
tmp/staging/sh4-linux/lib/libbz2.so.1.0.2
tmp/staging/sh4-linux/lib/libbz2.a
```

As well as being used during the stage task the staging related variables are used when building other packages. Looking at the gnupg recipe we see two bzip2 related items:

```
DEPENDS = "zlib bzip2"
...
EXTRA_OECONF = "--disable-ldap \
        --with-zlib=${STAGING_LIBDIR}/.. \
        --with-bzip2=${STAGING_LIBDIR}/.. \
        --disable-selinux-support"
```

Bzip2 is referred to in two places in the recipe:

DEPENDS

Remember that **DEPENDS** defines the list of build time dependencies. In this case the staged headers and libraries from bzip2 are required to build gnupg, and therefore we need to make sure

the bzip2 recipe has run and staged the headers and libraries. By adding the **DEPENDS** on bzip2 this ensures that this happens.

EXTRA_OECONF

This variable is used by the autotools class to provide options to the configure script of the package. In the gnupg case it needs to be told where the bzip2 headers and libraries are, and this is done via the *--with-bzip2* option. In this case it points to the directory which includes the lib and include subdirectories. Since OE doesn't define a variable for one level above the include and lib directories **..** is used to indicate one directory up. Without this, gnupg would search the host system headers and libraries instead of those we have provided in the staging area for the target.

Remember that staging is used to make things, such as headers and libraries, available for use by other recipes later on. While headers and libraries are the most common items requiring staging, other items such as the pkgconfig files need to be staged as well. For native packages, the binaries also need to be staged.

8.15. Autoconf: All about autotools

This section is to be completed:

- About building autoconf packages
- EXTRA_OECONF
- Problems with /usr/include, /usr/lib
- Configuring to search in the staging area
- -L${STAGING_LIBDIR} vs ${TARGET_LDFLAGS}
- Site files

8.16. Installation scripts: Running scripts during package installation and/or removal

Packaging systems such as .ipkg and .deb support pre and post installation and pre and post removal scripts which are run during package installation and/or package removal on the target system.

These scripts can be defined in your recipes to enable actions to be performed at the appropriate time. Common uses include starting new daemons on installation, stopping daemons during uninstall, creating new user and/or group entries during install, registering and unregistering alternative implementations of commands and registering the need for volatiles.

The following scripts are supported:

preinst

The preinst script is run prior to installing the contents of the package. During preinst the contents of the package are not available to be used as part of the script. The preinst scripts are not commonly used.

postinst

The postinst script is run after the installation of the package has completed. During postinst the contents of the package are available to be used. This is often used for the creation of volatile directories, registration of daemons, starting of daemons and fixing up of SUID binaries.

prerm

The prerm is run prior to the removal of the contents of a package. During prerm the contents of the package are still available for use by the script.

postrm

The postrm script is run after the completion of the removal of the contents of a package. During postrm the contents of the package no longer exist and therefore are not available for use by the script. Postrm is most commonly used for update alternatives (to tell the alternatives system that this alternative is not available and another should be selected).

Scripts are registered by defining a function for:

- pkg_<scriptname>_<packagename>

The following example from ndisc6 shows postinst scripts being registered for three of the packages that ndisc6 creates:

```
# Enable SUID bit for applications that need it
pkg_postinst_${PN}-rltraceroute6 () {
    chmod 4555 ${bindir}/rltraceroute6
}
pkg_postinst_${PN}-ndisc6 () {
    chmod 4555 ${bindir}/ndisc6
}
pkg_postinst_${PN}-rdisc6 () {
    chmod 4555 ${bindir}/rdisc6
}
```

Note: These scripts will be run via /**bin**/**sh** on the target device, which is typically the busybox sh but could also be bash or some other sh compatible shell. As always you should not use any bash extensions in your scripts and stick to basic sh syntax.

Note that several classes will also register scripts, and that any script you declare will have the script for the classes appended by these classes. The following classes all generate additional script contents:

update-rc.d

This class is used by daemons to register their init scripts with the init code.

Details are provided in the initscripts section.

module

This class is used by linux kernel modules. It's responsible for calling depmod and update-modules during kernel module installation and removal.

kernel

This class is used by the linux kernel itself. There is a lot of housekeeping required both when installing and removing a kernel and this class is responsible for generating the required scripts.

qpf

This class is used when installing and/or removing qpf fonts. It registers scripts to update the font paths and font cache information to ensure that the font information is kept up to date as fonts are installed and removed.

update-alternatives

This class is used by packages that contain binaries which may also be provided by other packages. It tells the system that another alternative is available for consideration. The alternatives system will create a symlink to the correct alternative from one or more available on the system.

Details are provided in the alternatives section.

gtk-icon-cache

This class is used by packages that add new gtk icons. It's responsible for updating the icon cache when packages are installed and removed.

gconf

package

The base class used by packaging classes such as those for .ipkg and .deb. The package class may create scripts used to update the dynamic linker's ld cache.

The following example from p3scan shows a postinst script which ensures that the required user and group entries exist, and registers the need for volatiles (directories and/or files under **/var**). In addition to explicitly declaring a postinst script it uses the update-rc.d class which will result in an additional entry being added to the postinst script to register the init scripts and start the daemon (via call to update-rc.d as described in the alternatives section).

```
inherit autotools update-rc.d
```

```
...

# Add havp's user and groups
pkg_postinst_${PN} () {
        grep -q mail: /etc/group || addgroup --system havp
        grep -q mail: /etc/passwd || \
            adduser --disabled-password --home=${localstatedir}/mail --system \
                    --ingroup mail --no-create-home -g "Mail" mail
        /etc/init.d/populate-volatile.sh update
}
```

Several scripts in existing recipes will be of the following form:

```
if [ x"$D" = "x" ]; then
    ...
fi
```

This is testing if the installation directory, **D**, is defined and if it is no actions are performed. The installation directory will not be defined under normal circumstances. The primary use of this test is to permit the application to be installed during root filesystem generation. In that situation the scripts cannot be run since the root filesystem is generated on the host system and not on the target. Any required script actions would need to be performed via an alternative method if the package is to be installed in the initial root filesystem (such as including any required users and groups in the default **passwd** and **group** files for example.)

8.17. Configuration files

Configuration files that are installed as part of a package require special handling. Without special handling as soon as the user upgrades to a new version of the package then changes they have made to the configuration files will be lost.

In order to prevent this from happening you need to tell the packaging system which files are configuration files. Such files will result in the user being asked how they want to handle any configuration file changes (if any), as shown in this example:

```
Downloading http://nynaeve.twibble.org/ipkg-titan-glibc//./p3scan_2.9.05d-r1_sh4.ipk
    Configuration file '/etc/p3scan/p3scan.conf'
    ==> File on system created by you or by a script.
    ==> File also in package provided by package maintainer.
       What would you like to do about it ?  Your options are:
        Y or I  : install the package maintainer's version
        N or O  : keep your currently-installed version
```

```
       D     : show the differences between the versions (if diff is installed)
 The default action is to keep your current version.
*** p3scan.conf (Y/I/N/O/D) [default=N] ?
```

To declare a file as a configuration file you need to define the **CONFFILES_<pkgname>** variable as a whitespace separated list of configuration files. The following example from clamav shows two files being marked as configuration files:

```
CONFFILES_${PN}-daemon = "${sysconfdir}/clamd.conf \
                          ${sysconfdir}/default/clamav-daemon"
```

Note the use of **${PN}-daemon** as the package name. The **${PN}** variable will expand to **clamav** and therefore these conf files are declared as being in the clamav-daemon package.

8.18. Package relationships

Explicit relationships between packages are support by packaging formats such as ipkg and deb. These relationships include describing conflicting packages and recommended packages.

The following variables control the package relationships in the recipes:

RRECOMMENDS

Used to specify other packages that are recommended to be installed when this package is installed. Generally this means while the recommended packages are not required they provide some sort of functionality which users would usually want.

RCONFLICTS

Used to specify other packages that conflict with this package. Two packages that conflict cannot be installed at the same time.

RREPLACES

Used to specify that the current package replaces an older package with a different name. During package installation the package that is being replaced will be removed since it is no longer needed when this package is installed.

RSUGGESTS

Used to provide a list of suggested packages to install. These are packages that are related to and useful for the current package but which are not actually required to use the package.

RPROVIDES

Used to explicitly specify what a package provides at runtime. For example hotplug support is provided by several packages, such as udev and linux-hotplug. Both declare that they runtime provide "hotplug". So any packages that require "hotplug" to work simply declare that it RDEPENDS on "hotplug". It's up to the distribution to specify which actual implementation of "virtual/hotplug" is used.

PROVIDES

Used to explicitly specify what a package provides at build time. This is typically used when two or more packages can provide the same functionality. For example there are several different X servers in OpenEmbedded, and each declared as providing "virtual/xserver". Therefore a package that depends on an X server to build can simply declare that it DEPENDS on "virtual/xserver". It's up to the distribution to specify which actual implementation of "virtual/xserver" is used.

8.19. Fakeroot: Dealing with the need for "root"

Sometimes packages require root permissions in order to perform some action, such as changing user or group owners or creating device nodes. Since OpenEmbedded will not keep the user and group information it's usually preferable to remove that from the makefiles. For device nodes it's usually preferable to create them from the initial device node lists or via udev configuration.

However if you can't get by without root permissions then you can use fakeroot to simulate a root environment, without the need to really give root access.

Using fakeroot is done by prefixing the task:

```
fakeroot do_install() {
```

Since this requires fakeroot you also need to add a dependency on **fakeroot-native**:

```
DEPENDS = "fakeroot-native"
```

See the fuse recipe for an example. Further information on fakeroot, including a description of how it works, is provided in the reference section: fakeroot.

8.20. Native: Packages for the build host

This section is to be completed.

- What native packages are
- Using require with the non-native package

8.21. Development: Strategies for developing recipes

This section is to be completed.

- How to go about developing recipes
- How to handle incrementally creating patches

- How to deal with site file issues
- Strategies for autotools issues

8.22. Advanced versioning: How to deal with rc and pre versions

Special care needs to be taken when specify the version number for rc and pre versions of packages.

Consider the case where we have an existing 1.5 version and there's a new 1.6-rc1 release that you want to add.

- 1.5: Existing version;
- 1.6-rc1: New version.

If the new package is given the version number 1.6-rc1 then everything will work fine initially. However when the final release happens it will be called 1.6. If you now create a 1.6 version of the package you'll find that the packages are sorted into the following order:

1. 1.5
2. 1.6
3. 1.6-rc1

This results in the packaging system, such as ipkg, considering the released version to be older than the rc version.

In OpenEmbedded the correct naming of pre and rc versions is to use the previous version number followed by a + followed by the new version number. So the 1.6-rc1 release would be given the version number:

- 1.5+1.6-rc1

These would result in the eventually ordering being:

1. 1.5
2. 1.5+1.6-rc1
3. 1.6

This is the correct order and the packaging system will now work as expected.

8.23. Require/include: Reusing recipe contents

In many packages where you are maintaining multiple versions you'll often end up with several recipes which are either identical, or have only minor differences between them.

The require and/or include directive can be used to include common content from one file into other. You should always look for a way to factor out common functionality into an include file when adding new versions of a recipe.

> **Note:** Both require and include perform the same function - including the contents of another file into this recipe. The difference is that require will generate an error if the file is not found while include will not. For this reason include should not be used in new recipes.

For example the clamav recipe looks like this:

```
require clamav.inc

PR = "r0"
```

Note that all of the functionality of the recipe is provided in the clamav.inc file, only the release number is defined in the recipe. Each of the recipes includes the same **clamav.inc** file to save having to duplicate any functionality. This also means that as new versions are released it's a simple matter of copying the recipe and resetting the release number back to zero.

The following example from iproute2 shows the recipe adding additional patches that are not specified by the common included file. These are patches only needed for newer release and by only adding them in this recipe it permits the common code to be used for both old and new recipes:

```
PR = "r1"

SRC_URI += "file://iproute2-2.6.15_no_strip.diff;striplevel=0 \
            file://new-flex-fix.patch"

require iproute2.inc

DATE = "060323"
```

The following example from cherokee shows a similar method of including additional patches for this version only. However it also shows another technique in which the configure task is defined in the recipe for this version, thus replacing the *configure* task that is provided by the common include:

```
PR = "r7"

SRC_URI_append = "file://configure.patch \
```

```
                  file://Makefile.in.patch \
                  file://Makefile.cget.patch \
                  file://util.patch"

require cherokee.inc

do_configure() {
        gnu-configize
        oe_runconf
        sed -i 's:-L\$:-L${STAGING_LIBDIR} -L\$:' ${S}/*libtool
}
```

8.24. Python: Advanced functionality with python

Recipes permit the use of python code in order to perform complex operations which are not possible with the normal recipe syntax and variables. Python can be used in both variable assignments and in the implementation of tasks.

For variable assignments python code is indicated via the use of *${@...}*, as shown in the following example:

```
TAG = ${@bb.data.getVar('PV',d,1).replace('.', '_')}
```

The above example retrieves the PV variable from the bitbake data object, then replaces any dots with underscores. Therefore if the **PV** was **0.9.0** then **TAG** will be set to **0_9_0**.

Some of the more common python code in use in existing recipes is shown in the following list:

bb.data.getVar(<var>,d,1)

Retrieve the data for the specified variable from the bitbake database for the current recipe.

<variable>.replace(<key>, <replacement>)

Find each instance of the key and replace it with the replacement value. This can also be used to remove part of a string by specifying **''** (two single quotes) as the replacement.

The following example would remove the **'-frename-registers'** option from the **CFLAGS** variable:

```
CFLAGS := "${@'${CFLAGS}'.replace('-frename-registers', '')}"
```

os.path.dirname(<filename>)

Return the directory only part of a filename.

This is most commonly seen in existing recipes when setting the **FILESDIR** variable (as described in the FILESPATH/FILESDIR section). By obtaining the name of the recipe file itself, **FILE**, and then using os.path.dirname to strip the filename part:

```
FILESDIR = "${@os.path.dirname(bb.data.getVar('FILE',d,1))}/make-${PV}"
```

Note however that this is no longer required as **FILE_DIRNAME** is automatically set to the dirname of the **FILE** variable and therefore this would be written in new recipes as:

```
FILESDIR = "$FILE_DIRNAME/make-${PV}"
```

<variable>.split(<key>)[<index>]

Splits the variable around the specified key. Use **[<index>]** to select one of the matching items from the array generated by the split command.

The following example from the recipe **genext2fs_1.3+1.4rc1.bb** would take the **PV** of **1.3+1.4rc1** and split it around the **+** sign, resulting in an array containing **1.3** and **1.4rc1**. It then uses the index of **[1]** to select the second item from the list (the first item is at index **0**). Therefore **TRIMMEDV** would be set to **1.4rc1** for this recipe:

```
TRIMMEDV = "${@bb.data.getVar('PV', d, 1).split('+')[1]}"
```

As well as directly calling built-in python functions, those functions defined by the existing classes may also be called. A set of common functions is provided by the base class in **classes/base.bbclass**:

base_conditional

This function is used to set a variable to one of two values based on the definition of a third variable. The general usage is:

```
${@base_conditional('<variable-name>', '<value>', '<true-result>', <false-result>', d)}'
```

where:

variable-name

This is the name of a variable to check.

value

This is the value to compare the variable against.

true-result

If the variable equals the value then this is what is returned by the function.

false-result

If the variable does not equal the value then this is what is returned by the function.

The following example from the openssl recipe shows the addition of either **-DL_ENDIAN** or **-DB_ENDIAN** depending on the value of **SITEINFO_ENDIANNESS** which is set to le for little endian targets and to be for big endian targets:

```
do_compile () {
    ...
    # Additional flag based on target endianess (see siteinfo.bbclass)
    CFLAG="${CFLAG} ${@base_conditional('SITEINFO_ENDIANNESS', 'le', '-DL_ENDIAN', '-DB_
    ...
```

base_contains

Similar to base_conditional except that it is checking for the value being an element of an array. The general usage is:

```
${@base_contains('<array-name>', '<value>', '<true-result>', <false-result>', d)}"
```

where:

array-name

This is the name of the array to search.

value

This is the value to check for in the array.

true-result

If the value is found in the array then this is what is returned by the function.

false-result

If the value is not found in the array then this is what is returned by the function.

The following example from the task-angstrom-x11 recipe shows base_contains being used to add a recipe to the runtime dependency list but only for machines which have a touchscreen:

```
RDEPENDS_angstrom-gpe-task-base := "\
    ...
    ${@base_contains("MACHINE_FEATURES", "touchscreen", "libgtkstylus", "",d)} \
    ...
```

Tasks may be implemented in python by prefixing the task function with "python ". In general this should not be needed and should be avoided where possible. The following example from the devshell recipe shows how the compile task is implemented in python:

```
python do_compile() {
    import os
    import os.path

    workdir = bb.data.getVar('WORKDIR', d, 1)
    shellfile = os.path.join(workdir, bb.data.expand("${TARGET_PREFIX}${DISTRO}-${MACHINE}-
```

```
    f = open(shellfile, "w")

    # emit variables and shell functions
        devshell_emit_env(f, d, False, ["die", "oe", "autotools_do_configure"])

    f.close()
}
```

8.25. Preferences: How to disable packages

When bitbake is asked to build a package and multiple versions of that package are available then bitbake will normally select the version that has the highest version number (where the version number is defined via the **PV** variable).

For example if we were to ask bitbake to build procps and the following packages are available:

```
$ ls recipes/procps
procps-3.1.15/    procps-3.2.1/    procps-3.2.5/    procps-3.2.7/    procps.inc
procps_3.1.15.bb  procps_3.2.1.bb  procps_3.2.5.bb  procps_3.2.7.bb
$
```

then we would expect it to select version **3.2.7** (the highest version number) to build.

Sometimes this is not actually what you want to happen though. Perhaps you have added a new version of the package that does not yet work or maybe the new version has no support for your target yet. Help is at hand since bitbake is not only looking at the version numbers to decided which version to build but it is also looking at the preference for each of those version. The preference is defined via the **DEFAULT_PREFERENCE** variable contained within the recipe.

The default preference (when no **DEFAULT_PREFERENCE** is specified) is zero. Bitbake will find the highest preference that is available and then for all the packages at the preference level it will select the package with the highest version. In general this means that adding a positive **DEFAULT_PREFERENCE** will cause the package to be preferred over other versions and a negative **DEFAULT_PREFERENCE** will cause all other packages to be preferred.

Imagine that you are adding procps version 4.0.0, but that it does not yet work. You could delete or rename your new recipe so you can build a working image, but what you really want to do is just ignore the new 4.0.0 version until it works. By adding:

```
DEFAULT_PREFERENCE = "-1"
```

to the recipe this is what will happen. Bitbake will now ignore this version (since all of the existing versions have a preference of 0). Note that you can still call bitbake directly on the recipe:

```
bitbake -b recipes/procps/procps_4.0.0.bb
```

This enables you to test, and fix the package manually without having bitbake automatically select it normally.

By using this feature in conjunction with overrides you can also disable (or select) specific versions based on the override. The following example from glibc shows that this version has been disabled for the sh3 architecture because it doesn't support sh3. This will force bitbake to try and select one of the other available versions of glibc instead:

```
recipes/glibc/glibc_2.3.2+cvs20040726.bb:DEFAULT_PREFERENCE_sh3 = "-99"
```

8.26. Initscripts: How to handle daemons

This section is to be completed.

- update-rc.d class
- sh syntax
- stop/stop/restart params
- sample/standard script?
- volatiles

8.27. Alternatives: How to handle the same command in multiple packages

Alternatives are used when the same command is provided by multiple packages. A classic example is busybox, which provides a whole set of commands such as **/bin/ls** and **/bin/find**, which are also provided by other packages such as coreutils (**/bin/ls**) and findutils (**/bin/find**).

A system for handling alternatives is required to allow the user to choose which version of the command they wish to have installed. It should be possible to install either one, or both, or remove one when both are installed etc, and to have no issues with the packages overwriting files from other packages.

The most common reason for alternatives is to reduce the size of the binaries. By cutting down on features, built in help, error messages and combining multiple binaries into one large binary it's possible to save considerable space. Often users are not expected to use the commands interactively in embedded appliances and therefore these changes have no visible effect to the user. In some situations users may have interactive access, or they may be more advanced users who want shell access on appliances that

normally don't provide it, and in these cases they should be able to install the full functional version if they desire.

8.27.1. Example of alternative commands

Most distributions include busybox in place of the full featured version of the commands. The following example shows a typical install in which the find command, which we'll use as an example here, is the busybox version:

```
root@titan:~$ find --version
find --version
BusyBox v1.2.1 (2006.12.17-05:10+0000) multi-call binary

Usage: find [PATH...] [EXPRESSION]

root@titan:~$ which find
which find
/usr/bin/find
```

If we now install the full version of find:

```
root@titan:~$ ipkg install findutils
ipkg install findutils
Installing findutils (4.2.29-r0) to root...
Downloading http://nynaeve.twibble.org/ipkg-titan-glibc//./findutils_4.2.29-r0_sh4.ipk
Configuring findutils

update-alternatives: Linking //usr/bin/find to find.findutils
update-alternatives: Linking //usr/bin/xargs to xargs.findutils
```

Then we see that the standard version of find changes to the full featured implementation:

```
root@titan:~$ find --version
find --version
GNU find version 4.2.29
Features enabled: D_TYPE O_NOFOLLOW(enabled) LEAF_OPTIMISATION
root@titan:~$ which find
which find
/usr/bin/find
```

8.27.2. Using update-alternatives

Two methods of using the alternatives system are available:

1. Via the update-alternatives class. This is the simplest method, but is not usable in all situations.
2. Via directly calling the update-alternatives command.

The update-alternatives class provides the simplest method of using alternatives but it only works for a single alternative. For multiple alternatives they need to be manually registered during post install.

Full details on both methods is provided in the update-alternatives class section of the reference manual.

8.28. Volatiles: How to handle the /var directory

The **/var** directory is for storing volatile information, that is information which is constantly changing and which in general may be easily recreated. In embedded applications it is often desirable that such files are not stored on disk or flash for various reasons including:

- The possibility of a reduced lifetime of the flash;
- The limited amount of storage space available;
- To ensure filesystem corruption cannot occur due to a sudden power loss.

For these reasons many of the OpenEmbedded distributions use a tmpfs based memory filesystem for **/var** instead of using a disk or flash based filesystem. The consequence of this is that all contents of the **/var** directory is lost when the device is powered off or restarted. Therefore special handling of **/var** is required in all packages. Even if your distribution does not use a tmpfs based **/var** you need to assume it does when creating packages to ensure the package can be used on those distributions that do use a tmpfs based **/var**. This special handling is provided via the **populate-volatiles.sh** script.

> **Note:** If your package requires any files, directories or symlinks in /**var** then it should be using the populate-volatiles facilities.

8.28.1. Declaring volatiles

This section is to be completed.

- how volatiles work
- default volatiles
- don’t include any /var stuff in packages
- even if your distro doesn’t use /var in tmpfs, others do
- updating the volatiles cache during install

8.28.2. Logging and log files

As a consequence of the volatile and/or small capacity of the **/var** file system some distributions choose methods of logging other than writing to a file. The most typical is the use of an in-memory circular log buffer which can be read using the **logread** command.

To ensure that each distribution is able to implement logging in a method that is suitable for its goals all packages should be configured by default to log via syslog, and not log directly to a file, if possible. If the distribution and/or end-user requires logging to a file then they can configure syslog and/or your application to implement this.

8.28.3. Summary

In summary the following are required when dealing with **/var**:

- Configure all logging to use syslog whenever possible. This leaves the decision on where to log up to the individual distributions.
- Don't include any **/var** directories, files or symlinks in packages. They would be lost on a reboot and so should not be included in packages.
- The only directories that you can assume exist are those listed in the default volatiles file: **recipes/initscripts/initscripts-1.0/volatiles**.
- For any other directories, files or links that are required in **/var** you should install your own volatiles list as part of the package.

8.29. Miscellaneous

This section is to be completed.

- about optimization
- about download directories
- about parallel builds
- about determining endianess (aka net-snmp, openssl, hping etc style)
- about PACKAGES_DYNAMIC
- about LEAD_SONAME
- about "python () {" - looks like it is always run when a recipe is parsed? see pam/libpam
- about SRCDATE with svn/cvs?
- about INHIBIT_DEFAULT_DEPS?
- about COMPATIBLE_MACHINE and COMPATIBLE_HOST

- about SUID binaries, and the need for postinst to fix them up
- about passwd and group (some comment in install scripts section already).

Chapter 9. Reference

9.1. autotools class

Autotools is one of the most commonly seen configuration methods for applications. Anything that uses the standard **./configure; make; make install** sequence is using autotools. Usually the configure script will support a large number of options to specify various installation directories, to disable and/or enable various features and options to specify search paths for headers and libraries.

The autotools class takes care of all of the details for you. It defines appropriate tasks for *configure*, *compile*, *stage* and *install*. At its simplest adding an inherit for the autotools class is all that is required. The netcat recipe for example is:

```
DESCRIPTION = "GNU Netcat"
HOMEPAGE = "http://netcat.sourceforge.net"
LICENSE = "GPLv2"
MAINTAINER = "Your name <yname@example.com>"
SECTION = "console/networking"
PR = "r1"

SRC_URI = "${SOURCEFORGE_MIRROR}/netcat/netcat-${PV}.tar.bz2"

inherit autotools
```

The header is defined, the location of the source code and then the inherit. For the simplest cases this is all that is required. If you need to pass additional parameters to the configure script, such as for enabling and/or disabling options, then they can be specified via the **EXTRA_OECONF** variable. This example from the lftp recipe shows several extra options being passed to the configure script:

```
EXTRA_OECONF = "--disable-largefile --disable-rpath --with-included-readline=no"
```

If you define your own tasks for *configure*, *compile*, *stage* or *install* (via **do_<taskname>**) then they will override the methods generated by the autotools class. If you need to perform additional operations (rather than replacing the generated operations) you can use the **do_<task>_append** or **do_<task>_prepend** methods. The following example from the conserver recipe shows some additional items being installed:

```
# Include the init script and default settings in the package
do_install_append () {
    install -m 0755 -d ${D}${sysconfdir}/default ${D}${sysconfdir}/init.d
    install -m 0644 ${WORKDIR}/conserver.default ${D}${sysconfdir}/default/conserver
    install -m 0755 ${WORKDIR}/conserver.init ${D}${sysconfdir}/init.d/conserver
}
```

9.1.1. oe_runconf / autotools_do_configure

Autotools generates a configuration method called **oe_runconf** which runs the actual configure script, and a method called **autotools_do_configure** which generates the configure file (runs automake and autoconf) and then calls **oe_runconf**. The generated method for the *configure* task, **do_configure**, just calls the **autotools_do_configure** method.

It is sometimes desirable to implement your own **do_configure** method, where additional configuration is required or where you wish to inhibit the running of automake and autoconf, and then manually call **oe_runconf**.

The following example from the ipacct recipe shows an example of avoiding the use of automake/autoconf:

```
do_configure() {
    oe_runconf
}
```

Sometimes manual manipulations of the autotools files is required prior to calling autoconf/automake. In this case you can defined your own **do_configure** method which performs the required actions and then calls **autotools_do_configure**.

9.1.2. Presetting autoconf variables (the site file)

The autotools configuration method has support for caching the results of tests. In the cross-compilation case it is sometimes necessary to prime the cache with per-calculated results (since tests designed to run on the target cannot be run when cross-compiling). These are defined via the site file(s) for the architecture you are using and may be specific to the package you are building.

Autoconf uses site files as definied in the **CONFIG_SITE** variable, which is a space seperate list of files to load in the specified order. Details on how this variable is set is provided in the siteinfo class (the class responsbile for setting the variable) section.

There are some things that you should keep in mind about the caching of configure tests:

1. Check the other site files to see if there any entries for the application you are attempting to build.

   Sometimes entries are only updated for the target that the developer has access to. If they exist for another target then it may provide a good idea of what needs to be defined.

2. Sometimes the same cache value is used by multiple applications.

This can have the side effect where a value added for one application breaks the build of another. It is a very good idea to empty the site file of all other values if you are having build problems to ensure that none of the existing values are causing problems.

3. Not all values can be stored in the cache

   Caching of variables is defined by the author of the configure script, so sometimes not all variables can be set via the cache. In this case it often means resorting to patching the original configure scripts to achieve the desired result.

All site files are shell scripts which are run by autoconf and therefore the syntax is the same as you would use in sh. There are two current methods of settings variables that is used in the existing site files. This include explicitly settings the value of the variable:

```
ac_cv_sys_restartable_syscalls=yes
```

and conditionally setting the value of the variable:

```
ac_cv_uchar=${ac_cv_uchar=no}
```

The conditional version is using shell syntax to say "*only set this to the specified value if it is not currently set*". The conditional version allows the variable to be set in the shell prior to calling configure and it will then not be replaced by the value from the site file.

Note: Site files are applied in order, so the application specific site files will be applied prior to the top level site file entries. The use of conditional assignment means that the first definition found will apply, while when not using conditionals the last definition found will apply.

It is possible to disable the use of the cached values from the site file by clearing the definition of **CONFIG_SITE** prior to running the configure script. Doing this will disable the use of the site file entirely. This however should be used as a last resort. The following example from the db recipe shows an example of this:

```
# Cancel the site stuff - it's set for db3 and destroys the
# configure.
CONFIG_SITE = ""
do_configure() {
    oe_runconf
}
```

9.2. binconfig class

The binconfig class is for packages that install **<pkg>-config** scripts that provide information about the build settings for the package. It is usually provided by libraries and then used by other packages to determine various compiler options.

Since the script is used at build time it is required to be copied into the staging area. All the actions performed by the class are appended to the *stage* task.

The actions performed by the binconfig class are:

1. Copies the **<x>-config** script from the package into **${STAGING_BINDIR}** directory;
2. If the package is not native then it modifies the contents of the **<x>-config** script in the staging area to ensure that all the paths in the script refer to the staging area;
3. If the package is native then the**<x>-config** script is renamed to **<x>-config-native** to ensure that the native and non-native versions do not interfere with each other.

A package is considered to be native if it also inherits the native class.

The class will search in source directory, **${S}**, and all its subdirectories, for files that end in **-config** and process them as described above. All that is required to use the class is the addition of binconfig in an inherit statement:

```
inherit autotools binconfig
```

9.3. Directories: Installation variables

The following table provides a list of the variables that are used to control the directories into which files are installed.

These variables can be used directly by the recipe to refer to paths that will be used after the package is installed. For example, when specifying the location of configuration files you need to specify the location on the target as show in the following example from quagga:

```
# Indicate that the default files are configuration files
CONFFILES_${PN} = "${sysconfdir}/default/quagga"
CONFFILES_${PN}-watchquagga = "${sysconfdir}/default/watchquagga"
```

When using these variables to actually install the components of a package from within a bitbake recipe they should used relative to the destination directory, **D**. The following example from the quagga recipe shows some addition files being manually installed from within the recipe itself:

```
do_install () {
        # Install init script and default settings
        install -m 0755 -d ${D}${sysconfdir}/default ${D}${sysconfdir}/init.d ${D}${sysconf
        install -m 0644 ${WORKDIR}/quagga.default ${D}${sysconfdir}/default/quagga
```

Variable name	Definition	Typical value
prefix	/usr	/usr
base_prefix	*(empty)*	*(empty)*
exec_prefix	${base_prefix}	*(empty)*
base_bindir	${base_prefix}/bin	/bin
base_sbindir	${base_prefix}/sbin	/sbin
base_libdir	${base_prefix}/lib	/lib
datadir	${prefix}/share	/usr/share
sysconfdir	/etc	/etc
localstatedir	/var	/var
infodir	${datadir}/info	/usr/share/info
mandir	${datadir}/man	/usr/share/man
docdir	${datadir}/doc	/usr/share/doc
servicedir	/srv	/srv
bindir	${exec_prefix}/bin	/usr/bin
sbindir	${exec_prefix}/sbin	/usr/sbin
libexecdir	${exec_prefix}/libexec	/usr/libexec
libdir	${exec_prefix}/lib	/usr/lib
includedir	${exec_prefix}/include	/usr/include
palmtopdir	${libdir}/opie	/usr/lib/opie
palmqtdir	${palmtopdir}	/usr/lib/opie

9.4. Directories: Staging variables

The following table provides a list of the variables that are used to control the directories into which files are staged.

Staging is used for headers, libraries and binaries that are generated by packages and are to be used in the generation of other packages. For example the libpcre recipe needs to make the include files and libraries for the target available on the host for other applications that depend on libpcre. So in addition to

packaging these files up for use in the binary package they are need to be installed in the staging are for use by other packages.

There are two common situations in which you will need to directly refer to the staging directories:

1. To specify where headers and libraries are to be found for libraries that your package depends on. In some cases these will be found automatically due to the default compiler settings used by OE, but in other cases you will need to explicitly tell your package to look in the staging area. This is more commonly needed with autoconf based packages that check for the presence of a specific package during the *configure* task.
2. In the *install_append* task for libraries to specify where to install the headers and libraries if not done automatically.

The following example from libpcre shows the installation of the libraries and headers from the package into the staging area. Note the use of the *oe_libinstall* helper function for installation of the libraries:

```
do_install_append () {
    install -d ${STAGING_BINDIR}
    install -m 0755 ${D}${bindir}/pcre-config ${STAGING_BINDIR}/
}
```

The following example from the flac recipe shows the location of the ogg libraries and included before explicitly passed to the configured script via EXTRA_OECONF so that it will correctly find ogg and enable support for it:

```
EXTRA_OECONF = "--disable-oggtest --disable-id3libtest \
        --with-ogg-libraries=${STAGING_LIBDIR} \
        --with-ogg-includes=${STAGING_INCDIR} \
        --without-xmms-prefix \
        --without-xmms-exec-prefix \
        --without-libiconv-prefix \
        --without-id3lib"
```

The following table lists the available variables for referring to the staging area:

Directory	Definition
STAGING_DIR	${TMPDIR}/sysroots
STAGING_BINDIR	${STAGING_DIR}/${HOST_SYS}/bin
STAGING_BINDIR_CROSS	${STAGING_DIR}/${BUILD_SYS}/bin/${HOST_SYS}
STAGING_BINDIR_NATIVE	${STAGING_DIR}/${BUILD_SYS}/bin
STAGING_LIBDIR	${STAGING_DIR}/${HOST_SYS}/lib
STAGING_INCDIR	${STAGING_DIR}/${HOST_SYS}/include
STAGING_DATADIR	${STAGING_DIR}/${HOST_SYS}/share

Directory	Definition
STAGING_LOADER_DIR	${STAGING_DIR}/${HOST_SYS}/loader
STAGING_FIRMWARE_DIR	${STAGING_DIR}/${HOST_SYS}/firmware
STAGING_PYDIR	${STAGING_DIR}/lib/python2.4
STAGING_KERNEL_DIR	${STAGING_DIR}/${HOST_SYS}/kernel
PKG_CONFIG_PATH	${STAGING_LIBDIR}/pkgconfig
QTDIR	${STAGING_DIR}/${HOST_SYS}/qt2
QPEDIR	${STAGING_DIR}/${HOST_SYS}
OPIEDIR	${STAGING_DIR}/${HOST_SYS}

9.5. distutils class

Distutils is a standard python system for building and installing modules. The *distutils* class is used to automate the building of python modules that use the distutils system.

Any python package that requires the standard python commands to build and install is using the distutils system and should be able to use this class:

```
python setup.py build
python setup.py install
```

The *distutils* class will perform the build and install actions on the **setup.py** provided by the package, as required for building distutils packages, including setting all the required parameters for cross compiling. It willl also perform the following actions:

1. Adds python-native to **DEPENDS** to ensure that python is built and installed on the build host. This also ensure that the version of python that is used during package creation matches the version of python that will be installed on the target.
2. Adds python-core to **RDEPENDS** to ensure that the python-core is installed when this module is installed. Note that you need to manually add any other python module dependencies to **RDEPENDS**.

The following example from the *moin* recipe shows how simple this can make a python package:

```
DESCRIPTION = "A full fledged WikiWiki system written in Python"
LICENSE = "GPL"
SECTION = "base"
PRIORITY = "optional"
MAINTAINER = "Your name <yname@example.com>"
```

```
PR = "r1"

SRC_URI = "${SOURCEFORGE_MIRROR}/moin/moin-${PV}.tar.gz"

inherit distutils
```

The header, source location and the inherit are all that is required.

9.6. fakeroot (device node handling)

The fakeroot program is designed to allow non-root users to perform actions that would normally require root privileges as part of the package generation process. It is used by the rootfs_ipkg class for root filesystem creation and by the image class for the creation of filesystem images. Some recipes also use fakeroot to assist with parts of the package installation (usually) or building where root privligeses are expected by the package.

In particular fakeroot deals with:

- Device nodes; and
- Ownership and group (uid & gid) management.

9.6.1. How fakeroot works

First of all we'll look at an example of how the fakeroot process works when used manually.

If we attempt to create a device node as a normal non-root user then the command will fail, telling is that we do not have permission to create device nodes:

```
~%> mknod hdc b 22 0
mknod: 'hdc': Operation not permitted
```

Yet the image class is able to create device nodes and include them in the final images, all without the need to have root privileges.

Let's try and create that node again, this time we'll run the commands from within a fakeroot process:

```
~%> ./tmp/staging/x86_64-linux/bin/fakeroot
~#> mknod hdc b 22 0
~#> ls -l hdc
brw-------  1 root root 22, 0 Aug 18 13:20 hdc
~#>
```

So it looks like we have successfully managed to create a device node, even though we did not have to give a password for the root user. In reality this device node still doesn't exist, it just looks like it exits.

Fakeroot is lying to the shell process and telling it that *"yes, this file exists and these are its properties"*. We'll talk more about how fakeroot actually works in a minute.

In this case **hdc** is the cd-rom drive, so let's try and actually mount the cd-rom:

```
~#> mkdir disk
~#> mount hdc disk
ERROR: ld.so: object 'libfakeroot.so.0' from LD_PRELOAD cannot be preloaded: ignored.
mount: only root can do that
~#>
```

So even though it appears we have root permissions, and that we created a device node, you see that the system gives an error about libfakeroot and about not being able to run mount because we are not root.

If we exit the fakeroot process and then look at the device node this is what we see:

```
~#> exit
~%> ls -l hdc
brw-------  1 user user 22, 0 Aug 18 13:20 hdc
~#>
```

Note that it isn't a device node at all, just an empty file owned by the current user!

So what exactly is fakeroot doing? It's using **LD_PRELOAD** to load a shared library into program which replaces calls into libc, such as open and stat, and then returns information to make it look like certain commands succeeded without actually performing those commands. So when creating a device node fakeroot will:

1. Intercept the mknod system call and instead of creating a device node it'll just create an empty file, owned by the user who run fakeroot;
2. It remembers the fact that mknod was called by root and it remembers the properties of the device node;
3. When a program, such as ls, calls stat on the file fakeroot remembers that it was device node, owned by root, and modifies that stat information to return this to ls. So ls sees a device node even though one doesn't exist.

When we tried to run mount we received the error **"ERROR: ld.so: object 'libfakeroot.so.0' from LD_PRELOAD cannot be preloaded: ignored."**. This is due to the fact that mount is an suid root binary, and for security reasons **LD_PRELOAD** is disabled on suid binaries.

There are some very important points to remember when dealing with fakeroot:

1. All information regarding devices nodes, uid and gids will be lost when fakeroot exists;

2. None of the device nodes, uids or gids will appear on disk. However if you tar up a directory from within fakeroot (for example), all of these device, uids and gids will appear correctly in the tar archive;
3. Any suid binaries will not interact with fakeroot;
4. Any static binaries will not interact with fakeroot;

9.6.2. Root filesystem, images and fakeroot

Many people have been confused by the generated root filesystem not containing any valid device nodes. This is in fact the expected behaviour.

When you look at a generated root filesystem you'll notice that the device nodes all appear to be incorrectly created:

```
~%> ls -l tmp/rootfs/dev | grep ttySC
-rw-r--r--  1 root root 0 Aug 16 13:07 ttySC0
-rw-r--r--  1 root root 0 Aug 16 13:07 ttySC1
-rw-r--r--  1 root root 0 Aug 16 13:07 ttySC2
~%>
```

These are empty files and not device nodes at all.

If we look in the image files generated from that root filesystem then everything is actually ok:

```
~%> tar -ztvf tmp/deploy/images/titan-titan-20060816030639.rootfs.tar.gz | grep " ./dev/tty
crw-r----- root/root     204,8 2006-08-16 13:07:12 ./dev/ttySC0
crw-r----- root/root     204,9 2006-08-16 13:07:12 ./dev/ttySC1
crw-r----- root/root    204,10 2006-08-16 13:07:12 ./dev/ttySC2
~%>
```

The images are created from within the same fakeroot process as the creation of the root filesystem and therefore it correctly picks up all of the special files and permissions from fakeroot.

NOTE: This means that you cannot use the root filesystem in tmp/rootfs directly on your target device. You need to use the .tar.gz image and uncompress it, as root, in order to generate a root filesystem which is suitable for use directly on the target (or as an NFS root).

9.6.3. Recipes and fakeroot

Some applications require that you have root permissions to run their installation routine, and this is another area where fakeroot can help. In a recipe the method for a standard task, such as the **do_install** method for the *install* task:

```
do_install() {
```

```
    install -d ${D}${bindir} ${D}${sbindir} ${D}${mandir}/man8 \
               ${D}${sysconfdir}/default \
               ${D}${sysconfdir}/init.d \
               ${D}${datadir}/arpwatch

    oe_runmake install DESTDIR=${D}
    oe_runmake install-man DESTDIR=${D}
    ...
```

can be modified to run within a fakeroot environment by prefixing the method name with fakeroot:

```
fakeroot do_install() {
    install -d ${D}${bindir} ${D}${sbindir} ${D}${mandir}/man8 \
               ${D}${sysconfdir}/default \
               ${D}${sysconfdir}/init.d \
               ${D}${datadir}/arpwatch

    oe_runmake install DESTDIR=${D}
    oe_runmake install-man DESTDIR=${D}
    ...
```

9.7. image class

The image class is used to generate filesystem images containing a root filesystem, as generated by the rootfs class for the package type, such as rootfs_ipkg class, for use on the target device. This could be a *jffs2* image which is to be written directly into the flash on the target device for example. In addition this class also configures the ipkg feeds (where to get updates from) and is able to generate multiple different image types.

Summary of the actions performed by the *image_ipkg* class:

1. Inherits the rootfs class for the appropriate package type, typically rootfs_ipkg class, in order to bring in the functionality required to generate a root filesystem image. The root filesystem image is generate from a set of of packages (typically .ipkg packages), and then the required images are generated using the contents of the root filesystem;
2. Sets **BUILD_ALL_DEPS = "1"** to force the dependency system to build all packages that are listed in the **RDEPENDS** and/or **RRECOMENDS** of the packages to be installed;
3. Determines the name of the image device tables or table (**IMAGE_DEVICE_TABLES/IMAGE_DEVICE_TABLE**) which will be used to describe the device nodes to create in **/dev** directory in the root filesystem;
4. Erases the contents of any existing root filesystem image, **${IMAGE_ROOTFS}**;

5. If devfs is not being used then the **/dev** directory, **${IMAGE_ROOTFS}/dev**, will be created and then populated with the device nodes described by the image device table or tables (using "**makedevs -r ${IMAGE_ROOTFS} -D <table>**" for each device table);
6. Calls into rootfs_ipkg class to install all of the required packages into the root filesystem;
7. Configures the ipkg feed information in the root filesystem (using **FEED_URIS** and **FEED_DEPLOYDIR_BASE_URI**);
8. Runs any image pre-processing commands as specified via **${IMAGE_PREPROCESS_COMMAND}**;
9. Calls **bbimage** on the root filesystem for each required image type, as specified via **${IMAGE_FSTYPES}**, to generate the actual filesystem images;
10. Runs any image post-processing commands, as specified via **${IMAGE_POSTPROCESS_COMMAND}**.

The following variables may be used to control some of the behaviour of this class (remember we use rootfs_ipkg class to build the filesystem image, so look at the variables defined by that class as well):

USE_DEVFS

Indicates if the image will be using devfs, the device filesystem, or not. If devfs is being used then no **/dev** directory will be required in the image. Set to **"1"** to indicate that devfs is being used. Note that devfs has been removed from the Linux kernel in the 2.6 series and most platforms are moving towards the use of udev as a replacement for devfs.

Default: **"0"**

IMAGE_DEVICE_TABLES

Specifies one, or more, files containing a list of the device nodes that should be created in the **/dev** directory of the image. Each file is searched for via the **${BBPATH}** and therefore can be specified as a file relative to the top of the build. Device files are processed in the specified order. NOTE: If **IMAGE_DEVICE_TABLE** is set then this variable is ignored.

Example: **IMAGE_DEVICE_TABLES = "files/device_table-minimal.txt files/device_table_add-sci.txt device_table_add-sm.txt"**

Default: **"files/device_table-minimal.txt"**

IMAGE_DEVICE_TABLE

Specifies the file that lists the device nodes that should be created in the **/dev** directory of the image. This needs to be an absolute filename and so should be specified relative to **${BBPATH}**. Only a single device table is supported. Use **IMAGE_DEVICE_TABLES** instead if you want to use multiple device tables.

Default: **""**

IMAGE_PREPROCESS_COMMAND

Additional commands to run prior to processing the image. Note that these command run within the same fakeroot instance as the rest of this class.

Default: **""**

IMAGE_POSTPROCESS_COMMAND

Additional commands to run after processing the image. Note that these command run within the same fakeroot instance as the rest of this class.

Default: **""**

IMAGE_FSTYPES

Specifies the type of image files to create. The supported image types, and details on modifying existing types and on creating new types, are described in the image types section. This variable is set to a space seperated list of image types to generate.

Example: **"jffs2 tar.gz"**

Default: **"jffs2"**

FEED_URIS

The name of the feeds to be configured in the image by default. Each entry consists of the feed name, followed by two pound signs and then followed by the actual feed URI.

Example: **FEED_URIS = "example##http://dist.example.com/ipkg-titan-glibc/"**

Default: **""**

FEED_DEPLOYDIR_BASE_URI

If set, configures local testing feeds using OE package deploy dir contents. The value is URL, corresponding to the ipk deploy dir.

Example: **FEED_DEPLOYDIR_BASE_URI = "http://192.168.2.200/bogofeed/"**

Default: ""

9.7.1. Special node handling (fakeroot)

Special nodes, such as **/dev** nodes, and files with special permissions, such as suid files, are handled via the fakeroot system. This means that when you view the contents of the root filesystem these device appear to be created incorrectly:

The **IMAGE_PREPROCESS_COMMAND** and **IMAGE_POSTPROCESS_COMMAND** variables will be processed within the same fakeroot instance as the rest of the rest of this class.

9.7.2. Device (/dev) nodes

There are two variables that can be defined for creating device nodes. The new method supports multiple device node tables and supports searching for these tables via the **${BBPATH}** so that relative file names may be used.

The following example from **machine/titan.conf** shows the use of multiple device tables:

```
# Add the SCI devices to minimal /dev
IMAGE_DEVICE_TABLES = "files/device_table-minimal.txt files/device_table_add-sci.txt device
```

It uses the standard minimal device tables but adds some additional items which are not normally needed:

files/device_table-minimal.txt

This is the standard minimal set of device nodes.

files/device_table_add-sci.txt

This contains details for creating the **/dev/SC{0,1,2}** nodes which are required for the SH processors on board SCI and SCIF serial ports. On the titan hardware the serial console is provided via one of these ports and so we require the device node to be present.

device_table_add-sm.txt

This contains details for creating the **/dev/sm0** and **/dev/sm0p{0,1,2}** devices nodes for the block driver, and the associated partitions, that are used to manage the on board flash on the titan hardware.

Prior to support for multiple device tables this would have required the creation of a titan specific device table.

9.7.3. Image types

The type of filesystem images to create are specified via the **IMAGE_FSTYPES** variable. A full description of the available image types, options of the images and details on creating new image types is provided in the image types section.

9.7.4. Package feeds

"Package feed", or feed for short, is a term used by **ipkg** package manager, commonly used in embedded systems, to name a package repository holding packages. Structurally, a feed is a directory - local, or on HTTP of FTP server, - holding packages and package descriptor file, named **Packages** or **Packages.gz** if compressed. Multiple feeds are supported.

OpenEmbedded has support to pre-configure feeds within generated images, so once image is installed on a device, user can immediately install new software, without the need to manually edit config files. There are several ways to pre-configure feed support, described below.

9.7.4.1. Method 1: Using existing feed

If you already have the feed(s) set up and available via specific URL, they can be added to the image using FEED_URIS variable:

```
FEED_URIS = " \
        base##http://oe.example.com/releases/${DISTRO_VERSION}/feed/base \
        updates##http://oe.example.com/releases/${DISTRO_VERSION}/feed/updates"
```

FEED_URIS contains list of feed descriptors, separated by spaces, per OE conventions. Each descriptor consists of feed name and feed URL, joined with "##". Feed name is an identifier used by ipkg to distinguish among the feeds. It can be arbitrary, just useful to the users to understood which feed is used for one or another action.

9.7.4.2. Method 2: Using OE deploy directory as a feed (development only)

OE internally maintains a feed-like collection of directories to create images from packages. This package deployment directory however has structure internal to OE and subject to change without notice. Thus, using it as feed directly is not recommended (distributions which ignored this recommendation are known to have their feeds broken when OE upgraded its internal mechanisms).

However, using deploy directory as feed directly may be beneficial during development and testing, as it allows developers to easily install newly built packages without many manual actions. To facilitate this, OE offers a way to prepare feed configs for using deploy dir as such. To start with this, you first need to configure local HTTP server to export a package deployment directory via HTTP. Suppose you will export it via URL "http://192.168.2.200/bogofeed" (where 192.168.2.200 is the address which will be reachable from the device). Add the following to your local.conf:

```
FEED_DEPLOYDIR_BASE_URI = "http://192.168.2.200/bogofeed"
```

Now you need to setup local HTTP server to actually export that directory. For Apache it can be:

```
Alias /bogofeed ${DEPLOY_DIR}

<Directory ${DEPLOY_DIR}>
    Options Indexes FollowSymLinks
    Order deny,allow
    Allow from 192.168.2.0/24
</Directory>
```

Replace ${DEPLOY_DIR} with the full path of deploy directory (last components of its path will be **deploy/ipk**).

Now, every image built will automatically contain feed configs for the deploy directory (as of time of writing, deploy directory is internally structured with per-arch subdirectories; so, there are several feed configs being generated, one for each subdirectory).

9.8. Image types

One of the most commonly used outputs from a build is a filesystem image containing the root filesystem for the target device. There are several variables which can be used to control the type of output images and the settings for those images, such as endianess or compression ratios. This section details the available images and the variables that effect them. See the image class section for details on how image generation is configured.

The final root file system will consist of all of the files located in image root filesystem directory, **${IMAGE_ROOTFS}**, which is usually **tmp/rootfs/${PN}** in the build area. One important difference between the images and the root file system directory is that any files which can only be created by privileged users, such as device nodes, will not appear in the **${IMAGE_ROOTFS}** directory but they will be present in any images that are generated. This is due to *fakeroot* system keeping track of these special files and making them available when generating the image - even though they do not appear in the root filesystem directory. For this reason it is important to always create an actual image to use for testing, even if it's just a **.tar** archive, to ensure you have the correct device nodes and any other special files.

9.8.1. Defining images

Each supported image type is defined via a set of variables. Each variables has the name of the image type appended to indicate the settings for that particular image type. The behaviour of the built in image types can be changed by modifying these variables, and new types can be created by defining these variables for the new type.

The variables that define an image type are:

IMAGE_CMD_<type>

Specifies the actual command that is run to generate an image of the specified type.

EXTRA_IMAGECMD_<type>

Used to pass additional command line arguments to the **IMAGE_CMD** without the need to redefine the entire image command. This is often used to pass options such as endianess and compression rations. You need to look at the **IMAGE_CMD** definition to determine how these options are being used.

IMAGE_ROOTFS_SIZE_<type>

For those image types that generate a fixed size image this variable is used to specify the required image size.

IMAGE_DEPENDS_<type>

Lists the packages that the **IMAGE_CMD** depends on. As an example the jffs2 filesystem creation depends on **mkfs.jffs2** command which is part of the mtd utilities and therefore depends on mtd-utils-native.

9.8.2. Available image types

The following image types are built in to OpenEmbedded:

jffs2

Creates jffs2 *"Journaling flash file system 2"* images. This is a read/write, compressed filesystem for mtd (flash) devices. It is not supported for block devices.

```
IMAGE_CMD_jffs2 = "mkfs.jffs2 \
  -x lzo \
  --root=${IMAGE_ROOTFS} \
  --faketime \
  --output=${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.jffs2 \
  ${EXTRA_IMAGECMD}"
```

The **EXTRA_IMAGECMD** variable for jffs2 passed to **mkfs.jffs2** and is left empty by default:

```
EXTRA_IMAGECMD_jffs2 = ""
```

This was not always empty, prior to 2007/05/02 the **EXTRA_IMAGECMD** variable for jffs2 was set to enable padding, to define the endianess and to specify the block size:

```
EXTRA_IMAGECMD_jffs2 = "--pad --little-endian --eraseblock=0x40000"
```

cramfs

Creates cramfs *"Compression ROM file system"* images. This is a read only compressed filesystem which is used directly by decompressing files into RAM as they are accessed. Files sizes are limited to 16MB, file system size is limited to 256MB, only 8-bit uids and gids are supported, no hard links are supported and no time stamps are supported.

```
IMAGE_CMD_cramfs = "mkcramfs ${IMAGE_ROOTFS} \
  ${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.cramfs \
  ${EXTRA_IMAGECMD}"
```

The **EXTRA_IMAGECMD** variable for cramfs is passed to **mkcramfs** and is left empty by default:

```
EXTRA_IMAGECMD_cramfs = ""
```

ext2

Creates an *"Extended Filesystem 2"* image file. This is the standard Linux non-journaling file system.

```
IMAGE_CMD_ext2 = "genext2fs -b ${IMAGE_ROOTFS_SIZE} \
  -d ${IMAGE_ROOTFS} \
  ${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.ext2 \
  ${EXTRA_IMAGECMD}"
```

The **EXTRA_IMAGECMD** variable for ext2 is passed to **genext2fs** and is left empty by default:

```
EXTRA_IMAGECMD_ext2 = ""
```

The **IMAGE_ROOTS_SIZE** variable is used to specify the size of the ext2 image and is set to 64k by default:

```
IMAGE_ROOTFS_SIZE_ext2 = "65536"
```

ext3

Creates an *"Extended Filesystem 3"* image file. This is the standard Linux journaling file system.

```
IMAGE_CMD_ext3 = "genext2fs -b ${IMAGE_ROOTFS_SIZE} \
  -d ${IMAGE_ROOTFS} \
  ${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.ext3 \
  ${EXTRA_IMAGECMD}; \
tune2fs -j ${DEPLOY_DIR_IMAGE}/tmp.gz/${IMAGE_NAME}.rootfs.ext3"
```

The **EXTRA_IMAGECMD** variable for ext3 is passed to **genext2fs** and is left empty by default:

```
EXTRA_IMAGECMD_ext3 = ""
```

The **IMAGE_ROOTS_SIZE** variable is used to specify the size of the ext3 image and is set to 64k by default:

```
IMAGE_ROOTFS_SIZE_ext3 = "65536"
```

ext2.gz

Creates a version of the ext2 filesystem image compressed with **gzip**.

```
IMAGE_CMD_ext2.gz = "rm -rf ${DEPLOY_DIR_IMAGE}/tmp.gz && \
mkdir ${DEPLOY_DIR_IMAGE}/tmp.gz; \
genext2fs -b ${IMAGE_ROOTFS_SIZE} -d ${IMAGE_ROOTFS} \
  ${DEPLOY_DIR_IMAGE}/tmp.gz/${IMAGE_NAME}.rootfs.ext2 \
  ${EXTRA_IMAGECMD}; \
gzip -f -9 ${DEPLOY_DIR_IMAGE}/tmp.gz/${IMAGE_NAME}.rootfs.ext2; \
mv ${DEPLOY_DIR_IMAGE}/tmp.gz/${IMAGE_NAME}.rootfs.ext2.gz \
  ${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.ext2.gz; \
rmdir ${DEPLOY_DIR_IMAGE}/tmp.gz"
```

The **EXTRA_IMAGECMD** variable for ext2.gz is passed to **genext2fs** and is left empty by default:

```
EXTRA_IMAGECMD_ext2.gz = ""
```

The **IMAGE_ROOTS_SIZE** variable is used to specify the size of the ext2 image and is set to 64k by default:

```
IMAGE_ROOTFS_SIZE_ext2.gz = "65536"
```

ext3.gz

Creates a version of the ext3 filesystem image compressed with **gzip**.

```
IMAGE_CMD_ext3.gz = "rm -rf ${DEPLOY_DIR_IMAGE}/tmp.gz && \
mkdir ${DEPLOY_DIR_IMAGE}/tmp.gz; \
genext2fs -b ${IMAGE_ROOTFS_SIZE} -d ${IMAGE_ROOTFS} \
  ${DEPLOY_DIR_IMAGE}/tmp.gz/${IMAGE_NAME}.rootfs.ext3 \
  ${EXTRA_IMAGECMD}; \
tune2fs -j ${DEPLOY_DIR_IMAGE}/tmp.gz/${IMAGE_NAME}.rootfs.ext3; \
gzip -f -9 ${DEPLOY_DIR_IMAGE}/tmp.gz/${IMAGE_NAME}.rootfs.ext3; \
mv ${DEPLOY_DIR_IMAGE}/tmp.gz/${IMAGE_NAME}.rootfs.ext3.gz \
  ${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.ext3.gz; \
rmdir ${DEPLOY_DIR_IMAGE}/tmp.gz"
```

The **EXTRA_IMAGECMD** variable for ext3.gz is passed to **genext2fs** and is left empty by default:

```
EXTRA_IMAGECMD_ext3.gz = ""
```

The **IMAGE_ROOTS_SIZE** variable is used to specify the size of the ext2 image and is set to 64k by default:

```
IMAGE_ROOTFS_SIZE_ext3.gz = "65536"
```

squashfs

Creates a squashfs image. This is a read only compressed filesystem which is used directly with files uncompressed into RAM as they are accessed. Files and filesystems may be up to 2^64 bytes in size, full 32-bit uids and gids are stored, it detects duplicate files and stores only a single copy, all meta-data is compressed and big and little endian filesystems can be mounted on any platform.

Squashfs uses gzip as its compression method.

```
IMAGE_CMD_squashfs = "mksquashfs ${IMAGE_ROOTFS} \
  ${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.squashfs \
  ${EXTRA_IMAGECMD} -noappend"
```

The **EXTRA_IMAGECMD** variable for squashfs is passed to **mksquashfs** and is left empty by default:

```
EXTRA_IMAGECMD_squashfs = ""
```

This was not always empty, prior to 2007/05/02 the **EXTRA_IMAGECMD** variable for squashfs specified the endianess and block size of the filesystem:

```
EXTRA_IMAGECMD_squashfs = "-le -b 16384"
```

squashfs-lzma

Creates a squashfs image using lzma compression instead of gzip which is the standard squashfs compression type. This is a read only compressed filesystem which is used directly with files uncompressed into RAM as they are accessed. Files and filesystems may be up to 2^64 bytes in size, full 32-bit uids and gids are stored, it detects duplicate files and stores only a single copy, all meta-data is compressed and big and little endian filesystems can be mounted on any platform.

Squashfs-lzma uses lzma as its compression method.

```
IMAGE_CMD_squashfs-lzma = "mksquashfs-lzma ${IMAGE_ROOTFS} \
  ${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.squashfs \
  ${EXTRA_IMAGECMD} -noappend"
```

The **EXTRA_IMAGECMD** variable for squashfs is passed to **mksquashfs-lzma** and is left empty by default:

```
EXTRA_IMAGECMD_squashfs-lzma = ""
```

This was not always empty, prior to 2007/05/02 the **EXTRA_IMAGECMD** variable for squashfs specified the endianess and block size of the filesystem:

```
EXTRA_IMAGECMD_squashfs-lzma = "-le -b 16384"
```

tar

Creates a .tar archive.

```
IMAGE_CMD_tar = "cd ${IMAGE_ROOTFS} && \
  tar -cvf ${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.tar ."
```

The **EXTRA_IMAGECMD** variable in not supported for tar images.

tar.gz

Creates a **gzip** compressed .tar archive.

```
IMAGE_CMD_tar.gz = "cd ${IMAGE_ROOTFS} && \
  tar -zcvf ${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.tar.gz ."
```

The **EXTRA_IMAGECMD** variable in not supported for .tar.gz images.

tar.bz2

Creates a **bzip2** compressed .tar archive.

```
IMAGE_CMD_tar.bz2 = "cd ${IMAGE_ROOTFS} && \
  tar -jcvf ${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.tar.bz2 ."
```

The **EXTRA_IMAGECMD** variable in not supported for tar.bz2 images.

cpio

Creates a .cpio archive:

```
IMAGE_CMD_cpio = "cd ${IMAGE_ROOTFS} && \
  (find . | cpio -o -H newc >${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.cpio)"
```

The **EXTRA_IMAGECMD** variable in not supported for cpio images.

cpio.gz

Creates a **gzip** compressed .cpio archive.

```
IMAGE_CMD_cpio.gz = cd ${IMAGE_ROOTFS} && \
  (find . | cpio -o -H newc | gzip -c -9 >${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.cpio
```

The **EXTRA_IMAGECMD** variable in not supported for cpio.gz images.

The above built in list of image types is defined in the bitbake configuration file: **org.openembedded.dev/conf/bitbake.conf**.

9.8.3. Custom image types

Custom image types can be created by defining the **IMAGE_CMD** variable, and optionally the **EXTRA_IMAGECMD**, **IMAGE_ROOTFS_SIZE** and **IMAGE_DEPENDS** variables, for your new image type.

An example can be found in **conf/machine/wrt54.conf** where it defines a new image type, *squashfs-lzma*, for a squashfs filesystem using lzma compression instead of the standard gzip compression (squashfs-lzma is now a standard type, but the example still serves to show the concept):

```
IMAGE_DEPENDS_squashfs-lzma = "squashfs-tools-native"
IMAGE_CMD_squashfs-lzma = "mksquashfs-lzma ${IMAGE_ROOTFS} \
  ${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.squashfs-lzma \
  ${EXTRA_IMAGECMD} -noappend"
EXTRA_IMAGECMD_squashfs-lzma = "-root-owned -le"
```

9.9. pkgconfig class

The pkgconfig class is for packages that install **<pkg>.pc** files. These files provide information about the build settings for the package which are then made available by the **pkg-config** command.

Since the contents of the **.pc** files are used at build time they need to be installed into the staging area. All the actions performed by this class are appended to the *stage* task.

The actions performed by the pkgconfig class are:

1. Copies the **<x>.pc** files into the **${PKG_CONFIG_PATH}** directory;
2. If the package is not native then it modifies the contents of the **<x>.pc** file in the **${PKG_CONFIG_PATH}** area to ensure that all the paths in the script refer to the staging area;

A package is considered to be native if it also inherits the native class.

The class will search the source directory, **${S}**, and all it's subdirectories, for files that end in **.pc** (it will ignore those that end in **-uninstalled.pc)** and process them as described above. All that is required to use the class is the addition of pkgconfig in an inherit statement:

```
inherit autotools pkgconfig
```

9.10. rootfs_ipkg class

The *rootf_ipk* class us used to create a root filesystem for the target device from a set of .ipkg packages. The end result is a directory containing all the files that need to be included in the root filesystem of the target device.

This class is normally not used directly, but instead used from the image class which creates images from a set of package (typically **.ipkg**) files.

Summary of actions performed by the *rootfs_ipkg* class:

1. Erase any existing root filesystem image by deleting the entire contents of **${IMAGE_ROOTFS}**;
2. Creates the device node directory, **${IMAGE_ROOTFS}/dev**;
3. Determines which packages to install in order to provide the locales that have been requested;
4. Configures ipkg to allow it to be used locally to install into the root filesystem **${IMAGE_ROOTFS}**;
5. Installs locale related .ipkg packages;
6. Installs the list of requested **.ipkg** packages, **${PACKAGE_INSTALL}**;
7. Creates ipkg's arch.conf as **${IMAGE_ROOTFS}/etc/ipkg/arch.conf**;
8. Runs any preinst and postinst scripts that were specified by the installed .ipkg packages;
9. Creates the system configuration directory **${IMAGE_ROOTFS}/${sysconfdir}** (that is the **/etc** directory on the target);
10. Runs and custom post-processing commands, as described by **${ROOTFS_POSTPROCESS_COMMAND}**;
11. Verifies that all the ipkg's were installed correctly and reports an error if they were not;

12. Makes available a set of functions which may be used by callers of the class: **zap_root_password**, **create_etc_timestamp** and **remove_init_link**;
13. Adds the rootfs task to run after the *install* task **"addtask rootfs before do_build and do_install"**.

The following variables may be used to control some of the behaviour of this class:

PACKAGE_INSTALL

The list of packages which will be installed into the root filesystem. This needs to be set in order for this class to perform any useful work.

Default: empty

ROOTFS_POSTPROCESS_COMMAND

Defines additional commands to run after processing of the root filesystem. Could be used to change roots password, remove parts of the install kernel such as the **zImage** kernel image or to edit the ipkg configuration for example.

Default: empty

PACKAGE_ARCHS

Defines the list of architectures that are support by the target platform. This is used to configure the arch settings for ipkg on the target system.

Default: **"all any noarch ${TARGET_ARCH} ${PACKAGE_EXTRA_ARCHS} ${MACHINE}"**

IMAGE_LINGUAS

Specifies which locales should be installed. This is often set to **""** to indicate that no locales will be installed.

Default: **"de-de fr-fr en-gb"**

EXTRA_IMAGEDEPENDS

A list of dependencies, this is appended to **DEPENDS**. This is typically used to ensure that any commands that are called by **ROOTFS_POSTPROCESS_COMMAND** are actually built by the system prior to being called.

Default: empty

BUILDNAME

The name of the build. This is either set by the distro configuration (for released versions) or set to a date stamp which is autogenerated by bitbake.

Default: **'date +%Y%m%d%H%M'**

IMAGE_ROOTFS

The path to the root of the filesystem image. You can use this when you need to explicitly refer to the root filesystem directory.

Default: **IMAGE_ROOTFS = "${TMPDIR}/rootfs/${PN}"**

DEPLOY_DIR

The base deploy dir. Used to find the directory containing the ipkg files.

Default: **DEPLOY_DIR = "${TMPDIR}/deploy"**

DEPLOY_DIR_IPK

The directory in which to search for the ipkg files that are to be installed in the root filesystem.

Default: **DEPLOY_DIR_IPK = "${DEPLOY_DIR}/ipk"**

Note that the entire process is run under the control of fakeroot in order to handle device files, uids and gids. The **ROOTFS_POSTPROCESS_COMMAND** is useful due to the fact that it runs within the same fakeroot instance as the rest of this class.

The class also provides a function **real_do_rootfs** which is executed without fakeroot and therefore can be used from other classes, such as image class, that are already running under the control of fakeroot.

9.11. SECTION variable: Package category

Sections are a means for categorising packages into related groups to enable users to find packages easier. The **SECTION** variable is used to declare which section a package belongs to. The most common use of the section information is in GUI based package management applications.

The default values for the section variables are:

- **SECTION = "base"**
- **SECTION_${PN}-doc = "doc"**
- **SECTION_${PN}-dev = "devel"**

Note that each package generated by a recipe can have it's own section and that by default documentation and development files are seperated out to their own sections.

The table of sections show the current usage of section information. This is a recomendation only, althought it is recomended that any additions or modifications be discusssed via the open embedded developer mailing list first.

Section	Description
admin	
base	Base system files. These are applications which are expected to be included as part of a base system and include things such as init scripts, core utilities, standard system daemons etc.
base/shell	Shells such as bash, tcsh, ksh etc.
bootloaders	Bootloaders, which are the applications responsible for loading the kernel from the appropriate location (disk, flash, network, etc.) and starting it running.
console	Applications which run on the console. These require no GUI related libraries or interfaces to run.
console/editors	
console/games	
console/multimedia	
console/network	
console/scientific	
console/telephony	
console/tools	
console/utils	
devel	Development related files. These include compilers, libraries, headers, debuggers etc.
devel/libs	
devel/perl	
devel/python	
devel/rexx	
devel/ruby	
devel/scheme	
devel/tcltk	
doc	Documentation, including man pages and sample configuration files.
e/apps	

e/libs	
e/utils	
fonts	Fonts that are not X11 or OPIE specific such as truetype fonts.
games	Games.
games/arcade	
gpe	GPE GUI enviroment. For the anything that provides or uses the GPE UI. Note that development and documentation related files should be in the appropriate devel and doc section, not under GPE.
gpe/applications	
gpe/base	
gpe/games	
gpe/libs	GPE runtime libraries. This does not include libraries used for development - they should be included in the appropriate devel section.
gpe/multimedia	
inputmethods	inputmethods that are neither libs, nor solely for GPE/Opie or the console
interpreters	
kde	KDE related applications.
kde/devel	
kernel	Linux kernels.
kernel/modules	Linux kernel modules. This include out-of-tree kernel modules.
kernel/userland	
libs	Runtime libraries. This does not include libraries used for development - they should be included in the appropriate devel section.
libs/inputmethods	
libs/multimedia	
libs/network	
network	
network/cms	
network/misc	
openmoko	Anything related to openmoko.org
openmoko/applications	
openmoko/base	
openmoko/examples	
openmoko/libs	
openmoko/pim	
openmoko/tools	
opie	OPIE GUI enviroment. For the anything that provides or uses the OPIE UI. Note that development and documentation related files should be in the appropriate devel and doc section, not under OPIE.
opie/applets	

opie/applications	
opie/base	
opie/codecs	
opie/datebook	
opie/decorations	
opie/fontfactories	
opie/fonts	OPIE specific fonts. General fonts, such as truetype fonts, should be in the fonts section.
opie/games	
opie/help	
opie/inputmethods	
opie/libs	OPIE runtime libraries. This does not include libraries used for development - they should be included in the appropriate devel section.
opie/multimedia	
opie/network	
opie/pim	
opie/security	
opie/settings	
opie/shell	
opie/styles	
opie/today	
utils	
x11	X11 GUI platform. For anything that provides or uses the X11 UI and is not GPE. Note that development and documentation related files should be in the appropriate devel and doc section, not under X11.
x11/applications	General applications.
x11/base	Core X11 applications.
x11/data	
x11/fonts	X11 specific fonts. General fonts, such as truetype fonts, should be in the fonts section.
x11/games	Games.
x11/gnome	Core gnome applications.
x11/gnome/libs	Gnome runtime libraries. This does not include libraries used for development - they should be included in the appropriate devel section.
x11/graphics	Applications which manipulate, display, edit, print etc. images, photos, diagrams etc.
x11/libs	X11 runtime libraries. This does not include libraries used for development - they should be included in the appropriate devel section.
x11/multimedia	Multimedia applications.
x11/network	
x11/office	Office and productivity applications.

x11/scientific	Scientific applications.
x11/utils	
x11/wm	Window managers.

The following tables lists some sections which may be in existing recipes. These should not be used in new recipes and should be renamed when updated existing recipes that use the specified sections.

Section	Action
apps	Replace with appropriate section
gui	Replace with appropriate section
media-gfx	Replace with appropriate section
multimedia	Replace with appropriate section
net	Replace with network
unknown	Replace with appropriate section
x11-misc	Replace with appropriate section

9.12. siteinfo class

The siteinfo class provides information for a target with a particular emphasis on determining the names of the site files to be passed to autoconf, as described in the autotools class. Full site information for your target can be determined by looking at the table in the class implementation found in the **classes/siteinfo.bbclass** file. A typical entry contains the name of the target and a list of site information for the target:

```
"sh4-linux":                "endian-little bit-32 common-glibc sh-common",
```

In the above example for sh4-linux target (that's a build for an sh4 processor using glibc) we see that the endianess and bit-size of target are defined and an additional set of site files that should be used are listed. These include a common site file for glibc and a common site file for sh processors (so sh3 and sh4 can share defines). A **"common"** entry is automatically added to the end of each of the definitions during processing.

The class makes available three variables based on the information provided for a target:

SITEINFO_ENDIANNESS

Defines the endianess of the target as either **"le"** (little endian) or **"be"** (big endian). The target must list either **endian-little** or **endian-big** in it's site information.

SITEINFO_BITS

Defines the bitsize of the target as either **"32"** or **"64"**. The target must list either **bit-32** or **bit-64** in it's site information.

CONFIG_SITE

Defines the site files to be used by autoconf. This is a space separated list of one or more site files for the target.

A typical use for the **SITEINFO_ENDIANNESS** and **SITEINFO_BITS** variables is to provide configuration within a recipe based on their values. The following example from the *openssl* recipe showw the correct define for the endiness of the target being passed to openssl via the compiler flags. The define to add to the flags is set based on the value of the **SITEINFO_ENDIANNESS** variable. Note that use of the *base_conditional* method (see the advanced python section) to select a value conditional on the endianess setting:

```
# Additional flag based on target endiness (see siteinfo.bbclass)
  CFLAG="${CFLAG} ${@base_conditional('SITEINFO_ENDIANNESS', 'le', '-DL_ENDIAN', '-DB_END
```

9.12.1. CONFIG_SITE: The autoconf site files

The autotools configuration method has support for caching the results of tests. In the cross-compilation case it is sometimes necessary to prime the cache with per-calculated results (since tests designed to run on the target cannot be run when cross-compiling). These are defined via the site file(s) for the architecture you are using and may be specific to the package you are building.

Which site files are used is determined via the **CONFIG_SITE** definition which is calculated via the siteinfo class. Typically the following site files will be checked for, and used in the order found:

endian-(big|little)

Either **endian-big** or **endian-little** depending on the endianess of the target. This site file would contain defines that only change based on if the target is little endian or big endian.

bit-(32|64)

Either **bit-32** or **bit-64** depending on the bitsize of the target. This site file would contain defines that only change based on if the target is a 32-bit or 64-bit cpu.

common-(libc|uclibc)

Either **common-libc** or **common-uclibc** based on the C library being used for the target. This site file would contain defines the are specific to the C library being used.

<arch>-common

A common site file for the target architecture. For i386, i485, i586 and i686 this would be **x86-common**, for sh3 and sh4 this would be **sh-common** and for various arm targets this would be **arm-common**.

common

This is a site file which is common for all targets and contains definitions which remain the same no matter what target is being built.

Each of the supported site files for a target is will be checked for in several different directories. Each time a file is found it as added to the list of files in the **CONFIG_SITE** variable. The following directories are checked:

org.openembedded.dev/recipes/<packagename>/site-<version>/

This directory is for site files which are specific to a particular version (where version is the PV of the package) of a package.

org.openembedded.dev/recipes/<packagename>/site/

This directory is for site files which are specific to a particular package, but apply to all versions of the package.

org.openembedded.dev/site/

This directory is for site files that are common to all packages. Originally this was the only site file directory that was supported.

9.13. SRC_URI variable: Source code and patches

All recipes need to contain a definition of **SRC_URI**. It determines what files and source code is needed and where that source code should be obtained from. This includes patches to be applied and basic files that are shipped as part of the meta-data for the package.

A typical **SRC_URI** contains a list of URL's, patches and files as shown in this example from quagga:

```
SRC_URI = "http://www.quagga.net/download/quagga-${PV}.tar.gz \
           file://ospfd-no-opaque-lsa-fix.patch \
           file://fix-for-lib-inpath.patch \
           file://quagga.init \
           file://quagga.default \
           file://watchquagga.init \
           file://watchquagga.default"
```

All source code and files will be placed into the work directory, **${WORKDIR}**, for the package. All patches will be placed into a **patches** subdirectory of the package source directory, **${S}**, and then automatically applied to the source.

Before downloading from a remote URI a check will be made to see if what is to be retrieved is already present in the download source directory, **${DL_DIR}**, along with an associated md5 sum. If the source is present in the downloaded sources directory and the md5 sum matches that listed in the associated md5 sum file, then that version will be used in preference to retrieving a new version . Any source that is retrieved from a remote URI will be stored in the download source directory and an appropriate md5 sum generated and stored alongside it.

Checksums for http/https/ftp/ftps uris are stored in each recipe in the form of

```
SRC_URI[md5sum] = "9a7a11ffd52d9c4553ea8c0134a6fa86"
SRC_URI[sha256sum] = "36bdb85c97b39ac604bc58cb7857ee08295242c78a12848ef8a31701921b9434"
```

for the first remote SRC_URI that has *no* explicit **name=foo** associated with it. Following *unnamed* SRC_URIs without a checksum will throw errors.

Each URI supports a set of additional options. These options are tag/value pairs of the form **"a=b"** and are semi-colon separated from each other and from the URI. The following example shows one option being included, the striplevel option:

```
file://ospfd-no-opaque-lsa-fix.patch;striplevel=2
```

The supported methods for fetching source and files are:

http, https, ftp, ftps

Used to download files and source code via the specified URL. These are fetched from the specified location using wget.

file

Used for files that are included locally in the meta-data. These may be plain files, such as init scripts to be added to the final package, or they may be patch files to be applied to other source.

cvs

Used to download from a CVS repository.

svn

Used to download from a subversion repository.

git

Used to download from a git repository.

When source code is specified as a part of **SRC_URI** it is unpacked into the work directory, **${WORKDIR}**. The unpacker recognises several archive and compression types and for these it will

decompress any compressed files and extract all of the files from archives into the work directory. The supported types are:

.tar

Tar archives which will be extracted with **"tar x --no-same-owner -f <srcfile>"**.

.tgz, .tar.gz, tar.Z

Gzip compressed tar archives which will be extracted with **"tar xz --no-same-owner -f <srcfile>"**.

.tbz, .tbz2, .tar.bz2

Bzip2 compressed tar archives which will be extracted with **"bzip2 -dc <srcfile> | tar x --no-same-owner -f -"**.

.gz, .Z, .z

Gzip compressed files which will be decompressed with **"gzip -dc <srcfile> > <dstfile>"**.

.bz2

Bzip2 compressed files which will be decompressed with **"bzip2 -dc <srcfile> > <dstfile>"**.

.xz

xz (LZMA2) compressed files which will be decompressed with **"xz -dc <srcfile> > <srcfile>"**.

.tar.xz

xz (LZMA2) compressed tar archives which will be decompressed with **"xz -dc <srcfile> | tar x --no-same-owner -f -"**.

.zip, .jar

Zip archives which will be extracted with **"unzip -q <srcfile>"**.

The downloading of the source files occurs in the *fetch* task, the unpacking and copying to the work directory occurs in the *unpack* task and the applying of patches occurs in the *patch* task.

9.13.1. http/https/ftp (wget)

The wget fetcher handles http, https and ftp URLs.

```
http://www.quagga.net/download/quagga-${PV}.tar.gz
```

Supported options:

md5sum

If an md5sum is provided then the downloaded files will only be considered valid if the md5sum of the downloaded file matches the md5sum option provided.

Related variables:

MIRRORS

Mirrors define alternative locations to download source files from. See the mirror section below for more information.

DL_DIR

The downloaded files will be placed in this directory with the name exactly as supplied via the URI.

9.13.2. file: for patches and additional files

The file URI's are used to copy files, included as part of the package meta data, into the work directory to be used when building the package. Typical use of the file URI's is to specify patches that be applied to the source and to provide additional files, such as init scripts, to be included in the final package.

The following example shows the specification of a patch file:

```
file://ospfd-no-opaque-lsa-fix.patch
```

Patch files are copied to the patches subdirectory of the source directory, **${S}/patches**, and then applied from the source directory. The patches are searched for along the path specified via the file path variable, **${FILESPATH},** and if not found the directory specified by the file directory variable, **${FILEDIR}**, is also checked.

The following example shows the specification of a non-patch file. In this case it's an init script:

```
file://quagga.init
```

Non-patch files are copied to the work directory, **${WORKDIR}**. You can access these files from within a recipe by referring to them relative to the work directory. The following example, from the quagga recipe, shows the above init script being included in the package by copying it during the *install* task:

```
do_install () {
    # Install init script and default settings
    install -m 0755 -d ${D}${sysconfdir}/default ${D}${sysconfdir}/init.d ${D}${sysconfdir}
    install -m 0644 ${WORKDIR}/quagga.init ${D}${sysconfdir}/init.d/quagga
...
```

Supported options:

apply={yes|no}

If set to 'yes' it is used as **"patch=1"** to define this file as a patch file. Patch files will be symlinked into **${S}/patches** and then applied to source from within the source directory, **${S}**. If set to 'no' the file will be copied to **${S}** during unpack.

striplevel

By default patches are applied with the **"-p 1"** parameter, which strips off the first directory of the pathname in the patches. This option is used to explicitly control the value passed to **"-p"**. The most typical use is when the patches are relative to the source directory already and need to be applied using **"-p 0"**, in which case the **"striplevel=0"** option is supplied.

9.13.3. cvs

The cvs fetcher is used to retrieve files from a CVS repository.

```
cvs://anonymous@cvs.sourceforge.net/cvsroot/linuxsh;module=linux;date=20051111
```

A cvs URI will retrieve the source from a cvs repository. Note that use of the *date=* to specify a checkout for specified date. It is preferable to use either a *date=* or a *tag=* option to select a specific date and/or tag from cvs rather than leave the checkout floating at the head revision.

Supported options:

module

The name of a module to retrieve. This is a required parameter and there is no default value.

tag

The name of a cvs tag to retrieve. Releases are often tagged with a specific name to allow easy access. Either a tag or a date can be specified, but not both.

date

The date to retrieve. This requests that files as of the specified date, rather then the current code or a tagged release. If no date or tag options are specified, then the date is set to the current date. The date is of any form accepted by cvs with the most common format being **"YYYYMMDD"**.

method

The method used to access the repository. Common options are **"pserver"** and **"ext"** (for cvs over rsh or ssh). The default is **"pserver"**.

rsh

The rsh command to use with the **"ext"** method. Common options are **"rsh"** or **"ssh"**. The default is **"rsh"**.

Related variables:

CVSDIR

The directory in which the cvs checkouts will be performed. The default is **${DL_DIR}/cvs**.

DL_DIR

A compressed tar archive of the retrieved files will be placed in this directory. The archive name will be of the form: **"<module>_<host>_<tag>_<date>.tar.gz"**. Path separators in **module** will be replaced with full stops.

9.13.4. svn

The svn fetcher is used to retrieve files from a subversion repository.

```
svn://svn.xiph.org/trunk;module=Tremor;rev=4573;proto=http
```

Supported options:

module

The name of a module to retrieve. This is a required parameter and there is no default value.

rev

The revision to retrieve. Revisions in subversion are integer values.

proto

The method to use to access the repository. Common options are **"svn"**, **"svn+ssh"**, **"http"** and **"https"**. The default is **"svn"**.

rsh

The rsh command to use with using the **"svn+ssh"** method. Common options are **"rsh"** or **"ssh"**. The default is **"ssh"**.

Related variables:

SVNDIR

The directory in which the svn checkouts will be performed.. The default is **${DL_DIR}/svn**.

DL_DIR

A compressed tar archive of the retrieved files will be placed in this directory. The archive name will be of the form: **"<module>_<host>_<path>_<revn>_<date>.tar.gz"**. Path separators in **path** and **module** will be replaced with full stops.

9.13.5. git

The git fetcher is used to retrieve files from a git repository.

```
SRC_URI = "git://www.denx.de/git/u-boot.git;protocol=git;tag=${TAG}"
```

Supported options:

branch

The git branch to retrieve from. The default is **"master"**.

tag

The git tag to retrieve. The default is **"master"**. This may be placed into **SRCREV** instead.

protocol

The method to use to access the repository. Common options are **"git"**, **"http"** and **"rsync"**. The default is **"rsync"**.

Related variables

SRCREV

The revision of the git source to be checked out. The default is **1** which is invalid and leads to an error.

GITDIR

The directory in which the git checkouts will be performed. The default is **${DL_DIR}/git**.

DL_DIR

A compressed tar archive of the retrieved files will be placed in this directory. The archive name will be of the form: **"git_<host><mpath>_<tag>.tar.gz"**. Path separators in **host** will be replaced with full stops.

9.13.6. Mirrors

The support for mirror sites enables spreading the load over sites and allows for downloads to occur even when one of the mirror sites are unavailable.

Default mirrors, along with their primary URL, include:

GNU_MIRROR

ftp://ftp.gnu.org/gnu

DEBIAN_MIRROR

ftp://ftp.debian.org/debian/pool

SOURCEFORGE_MIRROR

http://heanet.dl.sourceforge.net/sourceforge

GPE_MIRROR

http://handhelds.org/pub/projects/gpe/source

XLIBS_MIRROR

http://xlibs.freedesktop.org/release

XORG_MIRROR

http://xorg.freedesktop.org/releases

GNOME_MIRROR

http://ftp.gnome.org/pub/GNOME/sources

FREEBSD_MIRROR

ftp://ftp.freebsd.org/pub/FreeBSD

GENTOO_MIRROR

http://distfiles.gentoo.org/distfiles

APACHE_MIRROR

http://www.apache.org/dist

When creating new recipes this mirrors should be used when you wish to use one of the above sites by referring to the name of the mirror in the URI, as show in this example from flex:

```
SRC_URI = "${SOURCEFORGE_MIRROR}/lex/flex-2.5.31.tar.bz2
```

You can manually define your mirrors if you wish to force the use of a specific mirror by exporting the appropriate mirrors in **local.conf** with them set to the local mirror:

```
export GNU_MIRROR = "http://www.planetmirror.com/pub/gnu"
export DEBIAN_MIRROR = "http://mirror.optusnet.com.au/debian/pool"
export SOURCEFORGE_MIRROR = "http://optusnet.dl.sourceforge.net/sourceforge"
```

Mirrors can be extended in individual recipes via the use of **MIRRORS_prepend** or **MIRRORS_append**. Each entry in the list contains the mirror name on the left-hand side and the URI of the mirror on the right-hand side. The following example from libffi shows the addition of two URI for the **"${GNU_MIRROR}/gcc/"** URI:

```
MIRRORS_prepend () {
  ${GNU_MIRROR}/gcc/  http://gcc.get-software.com/releases/
  ${GNU_MIRROR}/gcc/  http://mirrors.rcn.net/pub/sourceware/gcc/releases/
}
```

9.13.7. Manipulating SRC_URI

Sometimes it is desirable to only include patches for a specific architecture and/or to include different files based on the architecture. This can be done via the **SRC_URI_append** and/or **SRC_URI_prepend** methods for adding additional URI's based on the architecture or machine name.

In this example from glibc, the patch creates a configuration file for glibc, which should only be used or the sh4 architecture. Therefore this patch is appended to the **SRC_URI**, but only for the sh4 architecture. For other architectures it is ignored:

```
# Build fails on sh4 unless no-z-defs is defined
SRC_URI_append_sh4 = " file://no-z-defs.patch;patch=1"
```

9.13.8. Source distribution (src_distribute_local)

In order to obtain a set of source files for a build you can use the *src_distribute_local* class. This will result in all the files that were actually used during a build being made available in a seperate directory and therefore they can be distributed with the binaries.

Enabling this option is as simple as activating the functionality by including the required class in one of your configuration files:

```
SRC_DIST_LOCAL = "copy"
INHERIT += "src_distribute_local"
```

Now during a build each recipe which has a LICENSE that mandates source availability, like the GPL, will be placed into the source distribution directory, **${SRC_DISTRIBUTEDIR}**, after building.

There are some options available to effect the option

SRC_DIST_LOCAL

Specifies if the source files should be copied, symlinked or moved and symlinked back. The default is **"move+symlink"**.

SRC_DISTRIBUTEDIR

Specifies the source distribution directory - this is why the source files that was used for the build are placed. The default is **"${DEPLOY_DIR}/sources"**.

The valid values for **SRC_DIST_LOCAL** are:

copy

Copies the files to the downloaded sources directory into the distribution directory.

symlink

Symlinks the files from the downloaded sources directory into the distribution directory.

move+symlink

Moves the files from the downloaded sources directory into the distribution directory. Then creates a symlink in the download sources directory to the moved files.

9.14. update-alternatives class

Some commands are available from multiple sources. As an example we have **/bin/sh** available from *busybox* and from *bash*. The busybox version is better from a size perspective, but limited in functionality, while the bash version is much larger but also provides far more features. The alternatives system is designed to handle the situation where two commands are provided by two, or more, packages. It ensures that one of the alternatives is always the currently selected one and ensures that there are no problems with installing and/or removing the various alternatives.

The update-alternatives class is used to register a command provided by a package that may have an alternative implementation in a some other package.

In the following sections we'll use the **/bin/ping** command as an example. This command is available as a basic version from busybox and as a more advanced version from iputils.

9.14.1. Naming of the alternative commands

When supplying alternative commands the target command itself is not installed directly by any of the available alternatives. This is to ensure that no package will replace files that were installed by one of the other available alternative packages. The alternatives system will create a symlink for the target command that points to the required alternative.

For the **/bin/ping** case this means that neither busybox nor iputils should actually install a command called **/bin/ping**. Instead we see that the iputils recipe installs it's version of ping as **/bin/ping.iputils**:

```
do_install () {
    install -m 0755 -d ${D}${base_bindir} ${D}${bindir} ${D}${mandir}/man8
    # SUID root programs
    install -m 4755 ping ${D}${base_bindir}/ping.${PN}
    ...
}
```

If you were to look at the busybox recipe you would see that it also doesn't install a command called **/bin/ping**, instead it installs it's command as **/bin/busybox**.

The important point to note is that neither package is installing an actual **/bin/ping** target command.

9.14.2. How alternatives work

Before proceeding lets take a look at how alternatives are handled. If we have a base image that includes only busybox then look at **/bin/ping** we see that it is a symlink to busybox:

```
root@titan:/etc# ls -l /bin/ping
lrwxrwxrwx    1 root     root            7 May  3  2006 /bin/ping -> busybox
```

This is what is expected since the busybox version of ping is the only one installed on the system. Note again that it is only a symlink and not an actual command.

If the iputils version of ping is now installed and we look at the **/bin/ping** command again we see that it has been changed to a symlink pointing at the iputils version of ping - **/bin/ping.iptils**:

```
root@titan:/etc# ipkg install iputils-ping
Installing iputils-ping (20020927-r2) to root...
Downloading http://nynaeve/ipkg-titan-glibc//iputils-ping_20020927-r2_sh4.ipk
Configuring iputils-ping
update-alternatives: Linking //bin/ping to ping.iputils
root@titan:/etc# ls -l /bin/ping
lrwxrwxrwx    1 root     root           12 May 13  2006 /bin/ping -> ping.iputils
```

The iputils version is considered to be the more fully featured version of ping and is therefore the default when both versions are installed.

What happens if the iputils-ping package is removed now? The symlink should be changed to point back at the busybox version:

```
root@titan:/etc# ipkg remove iputils-ping
Removing package iputils-ping from root...
update-alternatives: Linking //bin/ping to busybox
root@titan:/etc# ls -l /bin/ping
lrwxrwxrwx    1 root     root            7 May 13  2006 /bin/ping -> busybox
```

This simple example shows that the alternatives system is taking care of ensuring the symlink is pointing to the correct version of the command without any special interaction from the end users.

9.14.3. The update-alternatives command

Available alternatives need to be registered with the alternatives system. This is handled by the **update-alternatives** command. The help from the command shows it's usage options:

```
root@titan:/etc# update-alternatives --help
update-alternatives: help:

Usage: update-alternatives --install <link> <name> <path> <priority>
       update-alternatives --remove <name> <path>
       update-alternatives --help
<link> is the link pointing to the provided path (ie. /usr/bin/foo).
<name> is the name in /usr/lib/ipkg/alternatives/alternatives (ie. foo)
<path> is the name referred to (ie. /usr/bin/foo-extra-spiffy)
<priority> is an integer; options with higher numbers are chosen.
```

During postinst the update-alternatives command needs to be called with the install option and during postrm it needs to be called with the remove option.

The iputils recipe actual codes this directly (rather than using the class) so we can see an example of the command being called:

```
pkg_postinst_${PN}-ping () {
    update-alternatives --install ${base_bindir}/ping ping ping.${PN} 100
}
pkg_prerm_${PN}-ping () {
    update-alternatives --remove ping ping.${PN}
}
```

In both cases the name that the alternatives are registered against, **"ping"**, is passed in and the path to the iputils version of the command, **"ping.${PN}"**. For the install case the actual command name (where the symlink will be made from) and a priority value are also supplied.

9.14.4. Priority of the alternatives

So why did the alternatives system prefer the iputils version of ping over the busybox version? It's because of the relative priorities of the available alternatives. When iputils calls update-alternatives the last parameter passed is a priority:

```
 update-alternatives --install ${base_bindir}/ping ping ping.${PN} 100
```

So iputils is specifying a priority of 100 and if you look at busybox you'll see it specifies a priority of 50 for ping. The alternative with the highest priority value is the one that update-alternatives will select as the version to actual use. In this particular situation the authors have selected a higher priority for iputils since it is the more capable version of ping and would not normally be installed unless explicitly requested.

9.14.5. Tracking of the installed alternatives

You can actually see which alternatives are available and what their priority is on a target system. Here we have an example in which both busybox and iptuils-ping packages are installed:

```
root@titan:/etc# cat /usr/lib/ipkg/alternatives/ping
/bin/ping
busybox 50
ping.iputils 100
```

If we remove iputils-ping, then we see that alternatives file is updated to reflect this:

```
root@titan:/etc# cat /usr/lib/ipkg/alternatives/ping
/bin/ping
busybox 50
root@titan:/etc#
```

The file lists the command first, and then each of the available alternatives and their relative priorities.

9.14.6. Using the update-alternatives class

Neither busybox nor iputils actually use the update-alternatives class - they call the update-alternatives functions directly. They need to call the command directly since they need to register multiple alternatives and the class does not support this. The class can only be used when you have only a single alternative to register.

To use the class you need to inherent update-alternatives and then define the name, path, link and priority as show in the following example from the jamvm recipe:

```
inherit autotools update-alternatives

ALTERNATIVE_NAME = "java"
ALTERNATIVE_PATH = "${bindir}/jamvm"
ALTERNATIVE_LINK = "${bindir}/java"
ALTERNATIVE_PRIORITY = "10"
```

where the variables to be specified are:

ALTERNATIVE_NAME [Required]

The name that the alternative is registered against and needs to be the same for all alternatives registering this command.

ALTERNATIVE_PATH [Required]

The path of the installed alternative. (This was iputils.ping in the example used previously).

ALTERNATIVE_LINK [Optional]

The name of the actual command. This is what the symlink will be called and is the actual command that the use runs. The default value is: **"${bindir}/${ALTERNATIVE_NAME}"**

ALTERNATIVE_PRIORITY [Optional]

The priority of this alternative. The alternative with the highest valued priority will be selected as the default. The default value is: **"10"**.

The actual postinst and postrm commands that are registered by the class are:

```
update_alternatives_postinst() {
    update-alternatives --install ${ALTERNATIVE_LINK} ${ALTERNATIVE_NAME} ${ALTERNATIVE_PAT
}

update_alternatives_postrm() {
    update-alternatives --remove ${ALTERNATIVE_NAME} ${ALTERNATIVE_PATH}
}
```

9.15. update-rc.d class

Services which need to be started during boot need to be registered using the update-rc.d command. These services are required to have an init script which is installed into **/etc/init.d** that can be used to start and stop the service.

The following examples show a service being manually stopped and started using it's init script:

```
root@titan:/etc# /etc/init.d/syslog stop
Stopping syslogd/klogd: stopped syslogd (pid 1551).
stopped klogd (pid 1553).
done
root@titan:/etc# /etc/init.d/syslog start
Starting syslogd/klogd: done
root@titan:/etc#
```

The update-rc.d class takes care of the following automatically:

1. Registers the service with the system during postinst so it will be automatically started on boot;
2. Stops the service during prerm so it will no longer be running after being removed;
3. Unregisters the service during prerm so there will be no attempts to start the removed service during boot;
4. Adds a build and run time dependency on the update-rc.d package which it uses to register and unregister the services.

Usage is very simple, as shown by this example from dropbear:

```
INITSCRIPT_NAME = "dropbear"
INITSCRIPT_PARAMS = "defaults 10"
```

```
inherit autotools update-rc.d
```

where the variables are:

INITSCRIPT_NAME

The name of the init script, which the package will have installed into /etc/init.d

INITSCRIPT_PARAMS

The parameters to pass to the update-rc.d call during installation. Typically this will be the work default followed by either single number or a pair of numbers representing the start/stop sequence number (both are set to the same if only one number is supplied.)

The help from update-rc.d shows show the required parameters:

```
root@titan:/etc# update-rc.d -h
usage: update-rc.d [-n] [-f] [-r <root>] <basename> remove
       update-rc.d [-n] [-r <root>] [-s] <basename> defaults [NN | sNN kNN]
       update-rc.d [-n] [-r <root>] [-s] <basename> start|stop NN runlvl [runlvl] [...] .
                -n: not really
                -f: force
                -r: alternate root path (default is /)
                -s: invoke start methods if appropriate to current runlevel
root@titan:/etc#
```

The start and stop sequence numbers need to ensure that the the service is started at the appropriate time relative to other services, such as waiting for any service that it depends on before starting (networking for example). Unless the service is a system or security related service it's better to be started as late as possible.

9.15.1. Multiple update-rc.d packages

Defining multiple init scripts within the one recipe is also supported. Note that each init script must be in it's own package. The following example is from the quagga recipe:

```
# Main init script starts all deamons
# Seperate init script for watchquagga
INITSCRIPT_PACKAGES                = "${PN} ${PN}-watchquagga"
INITSCRIPT_NAME_${PN}              = "quagga"
INITSCRIPT_PARAMS_${PN}            = "defaults 15 85"
INITSCRIPT_NAME_${PN}-watchquagga  = "watchquagga"
INITSCRIPT_PARAMS_${PN}-watchquagga = "defaults 90 10"

inherit autotools update-rc.d
```

The variables that need to be declared are:

INITSCRIPT_PACKAGES

: The names of each package which includes an init script.

INITSCRIPT_NAME_x

: The same meaning as INITSCRIPT_NAME, but for the package x. This would be repeated for each package that includes an init script.

INITSCRIPT_PARAMS_x

: The same meaning as INITSCRIPT_PARAMS, but for the package x. This would be repeated for each package that includes an init script.