The Ubuntu Packaging Guide

Ubuntu Documentation Project <*ubuntu-doc@lists.ubuntu.com*>

The Ubuntu Packaging Guide

by Ubuntu Documentation Project <*ubuntu-doc@lists.ubuntu.com*>

Abstract

The Ubuntu Packaging Guide is an introduction to packaging programs for Ubuntu and other Debian based distributions

Credits and License

The following Ubuntu Documentation Team authors maintain this document:

- Jordan Mantha

The Ubuntu Packaging Guide is also based on the contributions of:

- Alexandre Vassalotti
- Jonathan Patrick Davies
- Ankur Kotwal
- Raphaël Pinson
- Daniel Chen
- Martin Pitt

Portions of the Ubuntu Packaging Guide are derived from the Debian New Maintainer's Guide and the Debian Policy Manual.

Disclaimer

Table of Contents

About This Guide

1. Conventions

The following notes will be used throughout the book:

A note presents interesting, sometimes technical, pieces of information related to the surrounding discussion.

A tip offers advice or an easier way of doing something.

A caution alerts the reader to potential problems and helps avoid them.

A warning advises the reader of a hazard that may arise in a given scenario.

Cross-reference conventions for print will be displayed as follows:

- Links to other documents or websites will look like *this* [http://www.ubuntu.com].

PDF, HTML, and XHTML versions of this document will use hyperlinks to handle cross-referencing.

Type conventions will be displayed as follows:

- File names or paths to directories will be shown in `monospace`.
- Commands that you type at a Terminal command prompt will be shown as:

  ```
  command to type
  ```

- Options that you click, select, or choose in a user interface will look like this.

Menu selections, mouse actions, and keyboard short-cuts:

- A sequence of menu selections will be displayed as follows: File → Open
- Mouse actions shall assume a right-handed mouse configuration. The terms “click” and “double-click” refer to using the left mouse button. The term “right-click” refers to using the right mouse button. The term “middle-click” refers to using the middle mouse button, pressing down on the scroll wheel, or pressing both the left and right buttons simultaneously, based on the design of your mouse.
- Keyboard shortcut combinations will be displayed as follows: **Ctrl**-**N** .Where the conventions for “Control”, “Shift,” and “Alternate” keys will be **Ctrl**, **Shift**, and **Alt**, respectively, and shall mean the first key is to be held down while pressing the second key.

2. Contributing and Feedback

This book is developed by the *Ubuntu Documentation Team* [https://wiki.ubuntu.com/DocumentationTeam]. *You* can contribute to this document by sending ideas or comments to the Ubuntu Documentation Team mailing list. Information about the team, its mailing lists, projects, etc. can be found on the *Ubuntu Documentation Team Website* [https://wiki.ubuntu.com/DocumentationTeam].

If you see a problem with this document, or would like to make a suggestion, you can simply file a bug report at the *Ubuntu Bugtracker* [https://launchpad.net/products/ubuntu-doc/+bugs]. Your help is vital to the success of our documentation!

Many thanks,

-Your Ubuntu Documentation Team

Chapter 1. Introduction

Welcome to the Ubuntu Packaging Guide! This guide is primarily addressed to those who would like to make and maintain Ubuntu packages. Although many of the concepts in this guide could be used to make binary packages for personal use, it is designed for those people wanting to distribute their packages to and for others. While it is also written with the Ubuntu Linux distribution in mind, it should also be useful for any Debian-based distribution.

There are several reasons you might want to learn how to package for Ubuntu. First, building and fixing Ubuntu packages is a great way to contribute to the Ubuntu community. It is also a good way to learn how Ubuntu and the applications you have installed work. Maybe you want to install a package that is not in the Ubuntu repositories. Hopefully after you have completed this guide you will have the tools and knowledge you need to do all of these things.

HTML and PDF versions of the manual are available online at *the Ubuntu Documentation website* [http://help.ubuntu.com].

You can buy this guide in book form from *our Lulu store* [http://www.lulu.com/ubuntu-doc]. You will only pay for the price of printing and postage.

1. Where to Begin

If you are completely new to Debian-based packaging then you will want to read this guide completely through, paying special attention to *Section 2, “Prerequisites” [p. 9]* , *Chapter 2, Getting Started [p. 11]* , and *Chapter 3, Basic Packaging [p. 16]* . People who are experienced with Debian-based packaging will find *Chapter 6, Ubuntu Packaging [p. 46]* and *Chapter 7, Bugs [p. 55]* most helpful.

2. Prerequisites

This guide assumes that the reader has a reasonable knowledge of building and installing software from source on Linux distributions. The guide also uses the Command Line Interface (CLI) throughout, so you should be comfortable using a terminal. You should be able to at least use the following:

- **make:** GNU Make is a very important software building tool. It is used to transform a complex compilation task into a trivial one. It is important that you know how to use it, because we will store most of the information about the packaging process in a Makefile. Documentation is available at the *GNU* [http://www.gnu.org/software/make/manual/make.html] website.
- **./configure:**This script is included in almost all Linux source, especially for software written in compiled languages such as C and C++. It is used to generate a Makefile (file used by make) that is properly configured for your system. Standard Debian packaging tools use it, so it is important that you know what the `configure` script does. Information on `./configure` can be found in the make documentation.
- **Apt/Dpkg:** Beyond the basic use of installing programs, apt and dpkg have many features that are useful for packaging.
  - **apt-cache dump** - lists every package in the cache. This command is especially helpful in combination with a grep pipe such as apt-cache dump | grep foo to search for packages whose names or dependencies include “foo”.
  - **apt-cache policy** - lists the repositories (main/restricted/universe/multiverse) in which a package exists.
  - **apt-cache show** - displays information about a binary package.
  - **apt-cache showsrc** - displays information about a source package.
  - **apt-cache rdepends** - shows reverse dependencies for a package (which packages require the queried one.
  - **dpkg -S** - lists the binary package to which a particular file belongs.
  - **dpkg -l** - lists currently installed packages. This is similar to apt-cache dump but for installed packages.
  - **dpkg -c** - lists the contents of a binary package. It is useful for ensuring that files are installed to the right places.
  - **dpkg -f** - shows the control file for a binary package. It is useful for ensuring that the dependencies are correct.
  - **grep-dctrl** - searches for specialized information in packages. It is a specific use of the grep package (but not installed by default).
- **diff and patch:** The diff program can be used to compare two files and to make patches. A typical example might be diff -ruN file.old file.new > file.diff. This command will create a diff (recursively if directories are used) that shows the changes, or “delta”, between the two files.

The patch program is used to apply a patch (usually created by diff or another similar program) to a file or directory. To apply the patch created above, we can invoke patch -p0 < file.diff. The *-p* tells patch how much it should strip from the paths for the file names in the patch. -p0 means to strip nothing, or leave the path intact.

Chapter 2. Getting Started

1. Binary and Source Packages

Most users of a Debian-based distribution such as Ubuntu will never have to deal with the actual source code that is used to create all of the applications on their computers. Instead, the source code is compiled into *binary* packages from the *source* package that contains both the source code itself and the rules for making the binary package. Packagers upload the source packages with their changes to the build systems that then compile the binary packages for each architecture. A separate system distributes the generated binary .deb files and source changes to the repository mirrors.

2. Packaging Tools

There are many tools written specifically for packaging on Debian-based systems. Many of them are not *essential* to creating packages but are very helpful and often automate repetitive tasks. Their man and info pages are good sources of information. However, the following is a list of packages that are deemed necessary to begin packaging:

build-essential
: is a metapackage that depends on libc6-dev, gcc, g++, make, and dpkg-dev. One package that you might not be familiar with is dpkg-dev. It contains tools such as dpkg-buildpackage and dpkg-source that are used to create, unpack, and build source and binary packages.

devscripts
: contains many scripts that make the packager's maintenance work much easier. Some of the more commonly used are debdiff, dch, debuild, and debsign.

debhelper and dh-make
: are scripts that automate common packaging tasks. dh-make can be used to do the initial "debianization" and provides many example files.

diff and patch
: are used to create and apply patches, respectively. They are used extensively in packaging because it is easier, cleaner, and more efficient to represent small changes as patches rather than to have multiple copies of a file.

gnupg
: is a complete and free replacement for PGP used to digitally sign files (including packages).

fakeroot
: simulates running a command with root privileges. This is useful for creating binary packages as a regular user.

lintian and linda
: dissect Debian packages and report bugs and Policy violations. They contain automated checks for many aspects of Debian Policy as well as for common errors.

pbuilder
: constructs a chroot system and builds a package inside the chroot. It is an ideal system to use to check that a package has correct build dependencies and to build clean packages to be tested and distributed.

3. The Personal Builder: pbuilder

Using pbuilder as a package builder allows you to build the package from within a chroot environment. You can build binary packages without using pbuilder, but you must have all the build dependencies installed on your system first. However, pbuilder allows the packager to check the build dependencies because the package is built within a minimal Ubuntu installation, and the build dependencies are downloaded according to the `debian/control` file.

The following is a brief guide to installing, using, and updating a pbuilder environment, however, there are many details of pbuilder usage that are outside the realm of this guide. The pbuilder man page has lots of information and should be consulted if you have problems or need more detailed information.

3.1. Installing and configuring a pbuilder environment

The first, and perhaps most obvious, thing to do is to install pbuilder. If you want to create a pbuilder for a release newer than the one you currently have installed, you will need to manually install the debootstrap .deb (from *http://packages.ubuntu.com*) from the newer release. To create a pbuilder execute:

```
sudo pbuilder create --distribution <distro> \
 --othermirror "deb http://archive.ubuntu.com/ubuntu <distro> universe multiverse"
```

where <distro> is the release you want (*edgy* for instance) to create the pbuilder for. If you would like to create more than one pbuilder environment you can append the *--basetgz* flag with the desired location for the compressed pbuilder environment. The default is `/var/cache/pbuilder/base.tgz`. If you do choose to use *--basetgz* you will need to use it with the other pbuilder commands so pbuilder knows which compressed build environment to use.

Creating a pbuilder environment will take some time as debootstrap essentially downloads a minimal Ubuntu installation.

A more flexible way to create a pbuilder (and perhaps multiple pbuilders) is to you a shell script.

3.2. Using the pbuilder

Now that you have a running pbuilder you can build binary packages from the source package by invoking:

```
sudo pbuilder build *.dsc
```

This will build all the source packages in the current directory. The resulting .debs and source packages can be found in `/var/cache/pbuilder/result/` (which can be changed with the *--buildresult* flag).

3.3. Updating the pbuilder

You should always have a current pbuilder whenever you are testing your source packages, especially when you are building for a development release that is rapidly changing, to ensure that the dependencies are properly found. To update your pbuilder, use:

```
sudo pbuilder update
```

If you would like to upgrade you pbuilder to a new release you can use pbuilder update in combination with the *--distribution* flag:

```
sudo pbuilder update --distribution <newdistro> --override-config
```

3.4. Multiple pbuilders

All of the information so far in this section on pbuilder has applied to having a single pbuilder. If you want to create more than one pbuilder you can create a shell script to handle the configuration for each pbuilder you want to create. An example of such a shell script can be found in `/usr/share/doc/pbuilder/examples/pbuilder-distribution.sh`. You can simply copy this example file somewhere in your path (putting it in `~/bin/` and adding this directory to your execution path is convenient) and then edit it according your needs. Normally you will need to only change DISTRIBUTION and add --othermirror as above. You can then call this script instead of pbuilder directly.

Chapter 3. Basic Packaging

Two of the problems that many novice packagers face are that there are multiple ways of packaging, and there is more than one tool to do the job. We will go through three examples with the common build systems. First, we will use no build helper. This approach is usually the most difficult and is not often used in practice but gives the most straightforward look at the packaging process. Second, we will use debhelper, the most common build system in Debian. It helps the packager by automating repetitive tasks. Third, we will briefly cover the **C**ommon **D**ebian **B**uild **S**ystem (CDBS), a more streamlined build system that uses debhelper.

Package development often requires installing many packages (especially -dev packages containing headers and other common development files) that are not part of a normal desktop Ubuntu installation. If you want to avoid installing extra packages or would like to develop for a different Ubuntu release (the development one, for instance) from what you currently have, the use of a chroot environment is highly recommended. A guide to setting up a *chroot [p. 62]* can be found in the Appendix.

1. Packaging From Scratch

Requirements: build-essential, automake, gnupg, lintian, fakeroot and *pbuilder [p. 14]*.

In this example we will be using the GNU *hello* [http://www.gnu.org/software/hello/hello.html] program as our example. You can download the source tarball from *ftp.gnu.org* [http://ftp.gnu.org/gnu/hello/hello-2.1.1.tar.gz]. For the purposes of this example, we will be using the `~/hello/` directory.

```
mkdir ~/hello
cd ~/hello
wget http://ftp.gnu.org/gnu/hello/hello-2.1.1.tar.gz
```

We will also compare our package to one that is already packaged in the Ubuntu repository. For now, we will place it in the `ubuntu` directory so we can look at it later. To get the source package, make sure you have a "deb-src" line in your `/etc/apt/sources.list` file for the Main repository. Then, simply execute:

```
mkdir ubuntu
cd ubuntu
apt-get source hello
cd ..
```

Unlike most apt-get commands, you do not need to have root privileges to get the source package, because it is downloaded to the current directory. In fact, it is recommended that you *only* use apt-get source as a regular user, because then you can edit files in the source package without needing root privileges.

What the apt-get source command does is:

1. Download the source package. A source package commonly contains a .dsc file describing the package and giving md5sums for the source package, an .orig.tar.gz file containing the source code from the author(s), and a .diff.gz file containing patches applied against the source code with the packaging information.
2. Untar the .orig.tar.gz file into the current directory.
3. Apply the gunzipped .diff.gz to the unpacked source directory.

If you manually download the source package (.dsc, .orig.tar.gz, and .diff.gz files), you can unpack them in the same way apt-get source does by using dpkg-source as follows:

```
dpkg-source -x *.dsc
```

The first thing you will need to do is make a copy of the original (sometimes called "upstream") tarball in the following format: `<packagename>_<version>.orig.tar.gz`. This step does two things.

First, it creates two copies of the source code. If you accidentally change or delete the working copy you can use the one you downloaded. Second, it is considered poor packaging practice to change the original source tarball unless absolutely necessary. See *Section 4, "Common Mistakes" [p. 34]* for reasons.

```
cp hello-2.1.1.tar.gz hello_2.1.1.orig.tar.gz
tar -xzvf hello_2.1.1.orig.tar.gz
```

The underscore, "_", between the package name (hello) and the version (2.1.1), as opposed to a hyphen, "-", is very important. Your source package will incorrectly be built as a Debian native package.

We now have a `hello-2.1.1` directory containing the source files. Now we need to create the customary debian directory where all the packaging information is stored, allowing us to separate the packaging files from the application source files.

```
mkdir hello-2.1.1/debian
cd hello-2.1.1/debian/
```

We now need to create the essential files for any Ubuntu source package: `changelog`, `control`, `copyright`, and `rules`. These are the files needed to create the binary packages (.deb files) from the original (upstream) source code. Let us look at each one in turn.

1.1. changelog

The `changelog` file is, as its name implies, a listing of the changes made in each version. It has a specific format that gives the package name, version, distribution, changes, and who made the changes at a given time. If you have a GPG key, make sure to use the same name and email address in `changelog` as you have in your key. The following is a template `changelog`:

```
package (version) distribution; urgency=urgency

  * change details
    more change details
  * even more change details

-- maintainer name <email address>[two spaces]  date
```

The format (especially of the date) is important. The date should be in RFC822 format, which can be obtained from the 822-date program.

Here is a sample `changelog` file for hello:

```
hello (2.1.1-1) edgy; urgency=low
```

```
   * New upstream release with lots of bug fixes.

-- Captain Packager <packager@coolness.com>  Wed,  5 Apr 2006 22:38:49 -0700
```

Notice that the version has a -1 appended to it, or what is called the Debian revision, which is used so that the packaging can be updated (to fix bugs for example) with new uploads within the same source release version.

Ubuntu and Debian have slightly different package versioning schemes to avoid conflicting packages with the same source version. If a Debian package has been changed in Ubuntu, it has *ubuntuX* (where *X* is the Ubuntu revision number) appended to the end of the Debian version. So if the Debian hello package was changed by Ubuntu, the version string would be `2.1.1-1ubuntu1`. If a package for the application does not exist in Debian, then the Debian revision is *0* (*e.g.*, `2.1.1-0ubuntu1`).

Now look at the `changelog` for the Ubuntu source package that we downloaded earlier:

```
less ../../ubuntu/hello-2.1.1/debian/changelog
```

Notice that in this case the *distribution* is *unstable* (a Debian branch), because the Debian package has not been changed by Ubuntu. Remember to set the *distribution* to your target distribution release.

At this point create a `changelog` file in the `debian` directory where you should still be.

1.2. control

The control file contains the information that the package manager (such as apt-get, synaptic, and aptitude) uses, build-time dependencies, maintainer information, and much more.

For the Ubuntu hello package, the control file looks something like:

```
Source: hello
Section: devel
Priority: optional
Maintainer: Captain Packager <packager@coolness.com>
Standards-Version: 3.6.1

Package: hello
Architecture: any
Depends: ${shlibs:Depends}
Description: The classic greeting, and a good example
 The GNU hello program produces a familiar, friendly greeting.  It
 allows non-programmers to use a classic computer science tool which
 would otherwise be unavailable to them.
 .
 Seriously, though: this is an example of how to do a Debian
 package.
 It is the Debian version of the GNU Project's `hello world' program
```

```
(which is itself an example for the GNU Project).
```

Create `control` using the information above (making sure to provide your information for the *Maintainer* field).

The first paragraph gives information about the source package. Let us go through each line:

- **Source:** This is the name of the source package, in this case, *hello*.
- **Section:** The apt repositories are split up into sections for ease of browsing and categorization of software. In this case, hello belongs in the *devel* section.
- **Priority:** This sets the importance of the package to users. It should be one of the following:
  - **Required** - packages that are essential for the system to work properly. If they are removed it is highly likely that your system will break in an unrecoverable way.
  - **Important** - minimal set of packages for a usable system. Removing these packages will not produce an unrecoverable breakage of your system, but they are generally considered important tools without which any Linux installation would be incomplete. Note: This does not include things like Emacs or even the X Window System.
  - **Standard** - Somewhat self explanatory.
  - **Optional** - in essence this category is for non-required packages, or the bulk of packages. However, these packages should not conflict with each other.
  - **Extra** - packages that may conflict with packages in one of the above categories. Also used for specialized packages that would only be useful to people who already know the purpose of the package.
- **Maintainer:** The package maintainer with email address.
- **Standards-Version:** The version of the *Debian Policy* [http://www.debian.org/doc/debian-policy/] to which the package adheres (in this case, version 3.6.1). An easy way to find the current version is *apt-cache show debian-policy | grep Version*.
- **Build-Depends:** One of the most important fields and often the source of bugs, this line lists the binary packages (with versions if necessary) that need to be installed in order to create the binary package(s) from the source package. Packages that are essential are required by *build-essential* and do not need to be included in the Build-Depends line. In the case of hello, all the needed packages are a part of build-essential, so a Build-Depends line is not needed. The list of build-essential packages can be found at `/usr/share/doc/build-essential/list`.

The second paragraph is for the binary package that will be built from the source. If multiple binary packages are built from the source package, there should be one section for *each* one. Again, let us go through each line:

- **Package:** The name for the binary package. Many times for simple programs (such as hello), the source and binary packages' names are identical.
- **Architecture:** The architectures for which the binary package(s) will be built. Examples are:
  - **all** - The source is *not* architecture-dependent. Programs that use Python or other interpreted languages would use this. The resulting binary package would end with `_all.deb`.

  - **any -** The source *is* architecture-dependent and should compile on all the supported architectures. There will be a .deb file for each architecture (`_i386.deb` for instance)
  - A subset of architectures (i386, amd64, ppc, etc.) can be listed to indicate that the source is architecture-dependent and does not work for all architectures supported by Ubuntu.
- **Depends:** The list of packages that the binary package depends on for functionality. For hello, we see `${shlibs:Depends}`, which is a variable that substitutes in the needed shared libraries. See the `dpkg-source` man page for more information.
- **Recommends:** Used for packages that are highly recommended and usually are installed with the package. Some package managers, most notably aptitude, automatically install Recommended packages.
- **Suggests:** Used for packages that are similar or useful when this package is installed.
- **Conflicts:** Used for packages that will conflict with this package. Both cannot be installed at the same time. If one is being installed, the other will be removed.
- **Description:** Both short and long descriptions are used by package managers. The format is:

  ```
  Description: <single line synopsis>
   <extended description over several lines>
  ```

  Note that there is one space at the beginning of each line in the long description.
  More information on how to make a good description can be found at *http://people.debian.org/~walters/descriptions.html*.

1.3. copyright

This file gives the copyright information. Generally, copyright information is found in the `COPYING` file in the program's source directory. This file should include such information as the names of the author and the packager, the URL from which the source came, a Copyright line with the year and copyright holder, and the text of the copyright itself. An example template would be:

```
This package was debianized by {Your Name} <your email address>
{Date}

It was downloaded from: {URL of webpage}

Upstream Author(s): {Name(s) and email address(es) of author(s)}

Copyright:
 Copyright (C) {Year(s)} by {Author(s)} {Email address(es)}

License:
```

As one can imagine, hello is released under the GPL license. In this case it is easiest to just copy the `copyright` file from the Ubuntu package:

```
cp ../../ubuntu/hello-2.1.1/debian/copyright .
```

You must include the complete copyright unless it is is GPL, LGPL, BSD, or Artistic License, in which case you can refer to the corresponding file in the `/usr/share/common-licenses/` directory.

Notice that the Ubuntu package's `copyright` includes a license statement for the manual. It is important that *all* the files in the source be covered by a license statement.

1.4. rules

The `rules` file is an executable Makefile that has rules for building the binary package from the source packages. For hello, it will be easier to use the `rules` from the Ubuntu package:

```
#!/usr/bin/make -f
# Sample debian/rules file - for GNU Hello.
# Copyright 1994,1995 by Ian Jackson.
# I hereby give you perpetual unlimited permission to copy,
# modify and relicense this file, provided that you do not remove
# my name from the file itself.  (I assert my moral right of
# paternity under the Copyright, Designs and Patents Act 1988.)
# This file may have to be extensively modified

package = hello
docdir = debian/tmp/usr/share/doc/$(package)

CC = gcc
CFLAGS = -g -Wall
INSTALL_PROGRAM = install

ifeq (,$(findstring noopt,$(DEB_BUILD_OPTIONS)))
  CFLAGS += -O2
endif
ifeq (,$(findstring nostrip,$(DEB_BUILD_OPTIONS)))
  INSTALL_PROGRAM += -s
endif

build:
        $(checkdir)
        ./configure --prefix=/usr
        $(MAKE) CC="$(CC)" CFLAGS="$(CFLAGS)"
        touch build

clean:
        $(checkdir)
        rm -f build
        -$(MAKE) -i distclean
        rm -rf *~ debian/tmp debian/*~ debian/files* debian/substvars

binary-indep:   checkroot build
        $(checkdir)
# There are no architecture-independent files to be uploaded
# generated by this package.  If there were any they would be
# made here.
```

```
binary-arch:    checkroot build
        $(checkdir)
        rm -rf debian/tmp
        install -d debian/tmp/DEBIAN $(docdir)
        install -m 755 debian/postinst debian/prerm debian/tmp/DEBIAN
        $(MAKE) INSTALL_PROGRAM="$(INSTALL_PROGRAM)" \
                prefix=$$(pwd)/debian/tmp/usr install
        cd debian/tmp && mv usr/info usr/man usr/share
        cp -a NEWS debian/copyright $(docdir)
        cp -a debian/changelog $(docdir)/changelog.Debian
        cp -a ChangeLog $(docdir)/changelog
        cd $(docdir) && gzip -9 changelog changelog.Debian
        gzip -r9 debian/tmp/usr/share/man
        gzip -9 debian/tmp/usr/share/info/*
        dpkg-shlibdeps debian/tmp/usr/bin/hello
        dpkg-gencontrol -isp
        chown -R root:root debian/tmp
        chmod -R u+w,go=rX debian/tmp
        dpkg --build debian/tmp ..

define checkdir
        test -f src/$(package).c -a -f debian/rules
endef

binary: binary-indep binary-arch

checkroot:
        $(checkdir)
        test $$(id -u) = 0

.PHONY: binary binary-arch binary-indep clean checkroot
```

Let us go through this file in some detail. One of the first parts you will see is the declaration of some variables:

```
package = hello
docdir = debian/tmp/usr/share/doc/$(package)

CC = gcc
CFLAGS = -g -Wall
INSTALL_PROGRAM = install

ifeq (,$(findstring noopt,$(DEB_BUILD_OPTIONS)))
  CFLAGS += -O2
endif
ifeq (,$(findstring nostrip,$(DEB_BUILD_OPTIONS)))
  INSTALL_PROGRAM += -s
endif
```

This section sets the CFLAGS for the compiler and also handles the `noopt` and `nostrip` DEB_BUILD_OPTIONS for debugging.

Next is the `build` rule:

```
build:
 $(checkdir)
 ./configure --prefix=/usr
 $(MAKE) CC="$(CC)" CFLAGS="$(CFLAGS)"
 touch build
```

This rule runs *./configure* with the proper prefix, runs *make*, and creates a `build` file that is a timestamp of the build to prevent erroneous multiple compilations.

The next rule is `clean`, which runs *make -i distclean* and removes the files that are made during the package building.

```
clean:
 $(checkdir)
 rm -f build
 -$(MAKE) -i distclean
 rm -rf *~ debian/tmp debian/*~ debian/files* debian/substvars
```

Next we see an empty `binary-indep` rule, because there are no architecture-independent files created in this package.

There are, however, many architecture-dependent files, so `binary-arch` is used:

```
binary-arch:    checkroot build
  $(checkdir)
  rm -rf debian/tmp
  install -d debian/tmp/DEBIAN $(docdir)
  install -m 755 debian/postinst debian/prerm debian/tmp/DEBIAN
  $(MAKE) INSTALL_PROGRAM="$(INSTALL_PROGRAM)" \
  prefix=$$(pwd)/debian/tmp/usr install
  cd debian/tmp && mv usr/info usr/man usr/share
  cp -a NEWS debian/copyright $(docdir)
  cp -a debian/changelog $(docdir)/changelog.Debian
  cp -a ChangeLog $(docdir)/changelog
  cd $(docdir) && gzip -9 changelog changelog.Debian
  gzip -r9 debian/tmp/usr/share/man
  gzip -9 debian/tmp/usr/share/info/*
  dpkg-shlibdeps debian/tmp/usr/bin/hello
  dpkg-gencontrol -isp
  chown -R root:root debian/tmp
  chmod -R u+w,go=rX debian/tmp
  dpkg --build debian/tmp ..
```

First, notice that this rule calls the `checkroot` rule to make sure the package is built as root and calls the `build` rule to compile the source. Then the `debian/tmp/DEBIAN` and `debian/tmp/usr/share/doc/hello` files are created, and the `postinst` and the `prerm>` scripts are installed to `debian/tmp/DEBIAN`. Then *make install* is run with a prefix that installs to the `debian/tmp/usr` directory. Afterward the documentation files (NEWS, ChangeLog, and the debian changelog) are gzipped and installed. *dpkg-shlibdeps* is invoked to find the shared library dependencies of the hello executable, and it stores the list in the `debian/substvars` file for the ${shlibs:Depends} variable in `control`. Then *dpkg-gencontrol* is run to create a control file for

the binary package, and it makes the substitutions created by *dpkg-shlibdeps*. Finally, after the permissions of the `debian/tmp` have been set, *dpkg --build* is run to build the binary .deb package and place it in the parent directory.

1.5. postinst and prerm

The `postinst` and `prerm` files are examples of maintainer scripts. They are shell scripts that are executed after installation and before removal, respectively, of the package. In the case of the Ubuntu hello package, they are used to install (and remove) the info file. Go ahead and copy them into the current `debian` directory.

```
cp ../../ubuntu/hello-2.1.1/debian/postinst .
cp ../../ubuntu/hello-2.1.1/debian/prerm .
```

1.6. Building the Source Package

Now that we have gone through the files in the `debian` directory for hello in detail, we can build the source (and binary) packages. First let us move into the root of the extracted source:

```
cd ..
```

Now we build the source package using dpkg-buildpackage:

```
dpkg-buildpackage -S -rfakeroot
```

The -S flag tells dpkg-buildpackage to build a source package, and the -r flag tells it to use fakeroot to allow us to have fake root privileges when making the package. dpkg-buildpackage will take the `.orig.tar.gz` file and produce a `.diff.gz` (the difference between the original tarball from the author and the directory we have created, `debian/` and its contents) and a `.dsc` file that has the description and md5sums for the source package. The `.dsc` and `*_source.changes` (used for uploading the source package) files are signed using your GPG key.

If you do not have a gpg key set up you will get an error from debuild. You can either set up a gpg key or use the *-us -uc* keys with debuild to turn off signing. However, you will not be able to have your packages uploaded to Ubuntu without signing them.

To make sure debuild finds the right gpg key you should set the DEBFULLNAME and DEBEMAIL environment variables (in your `~/.bashrc` for instance) to the name and email address you use for your gpg key and in the `debian/changelog`

Some people have reported that they were unable to get debuild to find their gpg key properly, even after setting the above environment variables. To get around this you can give debuild the *-k<keyid>* flag where <keyid> is your gpg key ID.

In addition to the source package, we can also build the binary package with pbuilder:

```
sudo pbuilder build ../*.dsc
```

Using pbuilder to build the binary packages is very important. It ensures that the build dependencies are correct, because pbuilder provides only a minimal environment, so all the build-time dependencies are determined by the `control` file.

We can check the source package for common mistakes using lintian:

```
cd ..
lintian -i *.dsc
```

2. Packaging with Debhelper

Requirements: The requirements from *Section 1, "Packaging From Scratch" [p. 17]* plus debhelper and dh-make

As a packager, you will rarely create packages from scratch as we have done in the previous section. As you can imagine, many of the tasks and information in the `rules` file, for instance, are common to packages. To make packaging easier and more efficient, you can use debhelper to help with these tasks. Debhelper is a set of Perl scripts (prefixed with *dh_*) that automate the process of package-building. With these scripts, building a Debian package becomes quite simple.

In this example, we will again build the GNU Hello package, but this time we will be comparing our work to the Ubuntu hello-debhelper package. Again, create a directory where you will be working:

```
mkdir ~/hello-debhelper
cd ~/hello-debhelper
wget http://ftp.gnu.org/gnu/hello/hello-2.1.1.tar.gz
mkdir ubuntu
cd ubuntu
```

Then, get the Ubuntu source package:

```
apt-get source hello-debhelper
cd ..
```

Like the previous example, the first thing we need to do is unpack the original (upstream) tarball.

```
tar -xzvf hello-2.1.1.tar.gz
```

Instead of copying the upstream tarball to `hello_2.1.1.orig.tar.gz` as we did in the previous example, we will let dh_make do the work for us. The only thing you have to do is rename the source folder so it is in the form of <*packagename*>-<*version*> where packagename is lowercase. In this case, just untarring the tarball produces a correctly named source directory so we can move into it:

```
cd hello-2.1.1
```

To create the initial "debianization" of the source we will use dh_make.

```
dh_make -e your.maintainer@address -f ../hello-2.1.1.tar.gz
```

dh_make will then ask you a series of questions:

```
Type of package: single binary, multiple binary, library, kernel module or cdbs?
[s/m/l/k/b] s

Maintainer name : Captain Packager
Email-Address   : packager@coolness.com
Date            : Thu,  6 Apr 2006 10:07:19 -0700
```

```
Package Name    : hello
Version         : 2.1.1
License         : blank
Type of Package : Single
Hit <enter> to confirm: Enter
```

Only run dh_make -e once. If you run it again after you do it the first time, it will not work properly. If you want to change it or made a mistake, remove the source directory and untar the upstream tarball afresh. Then you can migrate into the source directory and try again.

Running dh_make -e does two things:

1. Creates the `hello_2.1.1.orig.tar.gz` file in the parent directory,
2. Creates the basic files needed in `debian/` and many template files (.ex) that may be needed.

The Hello program is not very complicated, and as we have seen in *Section 1, "Packaging From Scratch" [p. 17]*, packaging it does not require much more than the basic files. Therefore, let us remove the `.ex` files:

```
cd debian
rm *.ex *.EX
```

For hello, you will also not need `README.Debian` (README file for specific Debian issues, not the program's README), `dirs` (used by dh_installdirs to create needed directories), `docs` (used by dh_installdocs to install program documentation), or `info` (used by dh_installinfo to install the info file) files into the `debian` directory. For more information on these files, see *Section 3, "dh_make example files" [p. 65]*.

At this point, you should have only `changelog`, `compat`, `control`, `copyright`, and `rules` files in the `debian` directory. From *Section 1, "Packaging From Scratch" [p. 17]*, the only file that is new is `compat`, which is a file that contains the debhelper version (in this case 4) that is used.

You will need to adjust the `changelog` slightly in this case to reflect that this package is named hello-debhelper rather than just hello:

```
hello-debhelper (2.1.1-1) edgy; urgency=low

  * Initial release

  -- Captain Packager <packager@coolness.com>  Thu,  6 Apr 2006 10:07:19 -0700
```

By using debhelper, the only things we need to change in `control` are the name (substituting hello for hello-debhelper) and adding debhelper (>= 4.0.0) to the *Build-Depends* field for the source package. The Ubuntu package for hello-debhelper looks like:

```
Source: hello-debhelper
Section: devel
Priority: extra
```

```
Maintainer: Capitan Packager <packager@coolness.com>
Standards-Version: 3.6.1
Build-Depends: debhelper (>= 4)

Package: hello-debhelper
Architecture: any
Depends: ${shlibs:Depends}
Conflicts: hello
Provides: hello
Replaces: hello
Description: The classic greeting, and a good example
 The GNU hello program produces a familiar, friendly greeting.  It
 allows non-programmers to use a classic computer science tool which
 would otherwise be unavailable to them.
 .
 Seriously, though: this is an example of how to do a Debian package.
 It is the Debian version of the GNU Project's `hello world' program
 (which is itself an example for the GNU Project).
 .
 This is the same as the hello package, except it uses debhelper to
 make the deb.  Please see debhelper as to what it is.
```

We can copy the `copyright` file and the `postinst` and `prerm` scripts from the Ubuntu hello-debhelper package, as they have not changed since *Section 1, "Packaging From Scratch" [p. 17]*. We will also copy the `rules` file so we can inspect it.

```
cp ../../ubuntu/hello-debhelper-2.1.1/debian/copyright .
cp ../../ubuntu/hello-debhelper-2.1.1/debian/postinst .
cp ../../ubuntu/hello-debhelper-2.1.1/debian/prerm .
cp ../../ubuntu/hello-debhelper-2.1.1/debian/rules .
```

The last file we need to look at is `rules`, where the power of debhelper scripts can be seen. The debhelper version of `rules` is somewhat smaller (54 lines as opposed to 72 lines in the version from *Section 1.4, "rules" [p. 22]*).

The debhelper version looks like:

```
 #!/usr/bin/make -f

package = hello-debhelper

CC = gcc
CFLAGS = -g -Wall

ifeq (,$(findstring noopt,$(DEB_BUILD_OPTIONS)))
  CFLAGS += -O2
endif

#export DH_VERBOSE=1
```

```
clean:
        dh_testdir
        dh_clean
        rm -f build
        -$(MAKE) -i distclean

install: build
        dh_clean
        dh_installdirs
        $(MAKE) prefix=$(CURDIR)/debian/$(package)/usr \
                mandir=$(CURDIR)/debian/$(package)/usr/share/man \
                infodir=$(CURDIR)/debian/$(package)/usr/share/info \
                install

build:
        ./configure --prefix=/usr
        $(MAKE) CC="$(CC)" CFLAGS="$(CFLAGS)"
        touch build

binary-indep: install
# There are no architecture-independent files to be uploaded
# generated by this package.  If there were any they would be
# made here.

binary-arch: install
        dh_testdir -a
        dh_testroot -a
        dh_installdocs -a NEWS
        dh_installchangelogs -a ChangeLog
        dh_strip -a
        dh_compress -a
        dh_fixperms -a
        dh_installdeb -a
        dh_shlibdeps -a
        dh_gencontrol -a
        dh_md5sums -a
        dh_builddeb -a

binary: binary-indep binary-arch

.PHONY: binary binary-arch binary-indep clean checkroot
```

Notice that tasks like testing if you are in the right directory (dh_testdir), making sure you are building the package with root privileges (dh_testroot), installing documentation (dh_installdocs and dh_installchangelogs), and cleaning up after the build (dh_clean) are handled automatically. Many packages much more complicated than hello have `rules` files no bigger because the debhelper scripts handle most of the tasks. For a complete list of debhelper scripts, please see *Section 4, “List of debhelper scripts” [p. 67]* . They are also well documented in their respective man pages. It is a useful exercise to read the man page (they are well written and not lengthy) for each helper script used in the above `rules` file.

2.1. Building the Source Package

Now that we have gone through the files in the `debian` directory for hello-debhelper, we can build the source (and binary) packages. First, let us move back into the source directory:

```
cd ..
```

Now we build the source package using debuild, a wrapper script for dpkg-buildpackage:

```
debuild -S
```

the binary package, using pbuilder:

```
sudo pbuilder build ../*.dsc
```

and finally check the source package for common mistakes using lintian:

```
cd ..
lintian -i *.dsc
```

3. Packaging With CDBS

CDBS is a tool that uses debhelper to make building and maintaining Debian packages even easier. It has many advantages:

- It produces a short, readable, and efficient `debian/rules`
- It automates debhelper and autotools for you, so you do not have to worry about repetitive tasks
- It helps you focus on more important packaging problems, because it helps without limiting customization
- Its classes have been well tested, so you can avoid dirty hacks to solve common problems
- Switching to CDBS is easy
- It is extensible

3.1. Using CDBS in packages

Using CDBS for Ubuntu packages is very easy. After adding cdbs to the Build-Depends in `debian/control`, a basic `debian/rules` file using CDBS can fit in 2 lines. For a simple C/C++ application with no extra rules, such as hello, `debian/rules` can look like this :

```
#!/usr/bin/make -f

include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/class/autotools.mk
```

That is all you need to build the program! CDBS handles installing and cleaning. You can then use the `.install` and `.info` files to tune your package with the usual debhelper functions in the various sections for `debian/rules`.

Do not use DEB_AUTO_UPDATE_DEBIAN_CONTROL:=yes to automatically change `debian/control`. It can cause bad things, and Debian considers it a reason to reject a package from entering the archives. See *http://ftp-master.debian.org/REJECT-FAQ.html* [http://ftp-master.debian.org/REJECT-FAQ.html] for more information.

As you can see, CDBS mostly works by including `.mk` Makefiles in `debian/rules`. The cdbs package provides such files in `/usr/share/cdbs/1/` that allow you to do quite a lot of packaging tasks. Other packages, such as quilt, add modules to CDBS and can be used as Build-Depends. Note that you can also use your own CDBS rules and include them in the package. The most useful modules included with the cdbs package are:

- `rules/debhelper.mk`: Calls debhelper in all required sections
- `rules/dpatch.mk`: Allows you to use dpatch to ease patching the source
- `rules/simple-patchsys.mk`: Provides a very easy way to patch the source
- `rules/tarball.mk`: Allows you to build packages using the compressed tarball in the package
- `class/autotools.mk`: Calls autotools in all required sections

- `class/gnome.mk`: Builds GNOME programs (requires the proper Build-Depends in `debian/control`)
- `class/kde.mk`: Builds KDE programs (requires the proper Build-Depends in `debian/control`)
- `class/python-distutils.mk`: Facilitates packaging Python programs

3.2. More information on CDBS

For more information on CDBS, see Marc Dequènes's guide at *https://perso.duckcorp.org/duck/cdbs-doc/cdbs-doc.xhtml*.

4. Common Mistakes

4.1. dh_make Example Files

When you use dh_make to create the initial "debianization", example files for various tasks are created in the `debian/` directory. The templates have a .ex extension. If you want to use one, rename it to remove the extension. If you do not need it, remove it to keep the `debian/` directory clean.

4.2. Changing the Original Tarball

There are two types of source packages, native and non-native. A native package is one that is specific to Ubuntu/Debian. It has the debian/ directory containing the packaging information and any changes to the source included in the tarball (usually <packagename>_<version>.tar.gz). Non-native packages are more common. A non-native package splits the source package into a <packagename>_<version>.orig.tar.gz tarball that is identical (hopefully including md5sum) to the source tarball downloaded from the project's homepage and a .diff.gz file that contains all the differences (`debian/` directory and patches) from the original source tarball.

Here is a list of potential problems that can occur if you change the original tarball:

1. Reproducibility

   If you take just the .diff.gz and .dsc, you or someone else has no means to reproduce the changes in the original tarball.

2. Upgradeability

   It is much easier to upgrade to a new upstream (from the author) version if the .orig.tar.gz is preserved and there is a clear separation between the upstream source and the changes made to produce the Ubuntu source package.

3. Debian to Ubuntu Synchronization

   Changing original tarballs makes it hard to automatically sync from Debian to Ubuntu. Normally, only the .diff.gz and .dsc files change within the same upstream version, since the .orig.tar.gz file is shared by all the Debian or Ubuntu revisions. It is much more difficult to sync if the md5sums of the .orig.tar.gz files are not the same.

4. Usage of Revision Control for Debian package

   If you use svn (svn-buildpackage) to handle your Debian package, you usually don't store the original tarball inside. If someone else does a checkout, he'll need to get the original tarball separately. Other revision control systems can be used to track only the packaging files (`debian/`, etc.) and not the whole source. However, if the .orig.tar.gz is not the same, then obviously problems can occur.

5. Security tracking

Consider a situation where someone *wants* to introduce a backdoor/rootkit or other evil stuff. If the original tarball is intact, it can be scanned easily through the .diff.gz to see if the person who modified the package tried to do something evil. If the tarball has changed, however, you also need to check the differences between the tarball and the original source.

You still have to trust the authors of the software not to do anything evil, but that is the case regardless of whether the original is changed.

6. The .diff.gz

   The option to use the .diff.gz to reflect changes to the original tarball already exists, so it is easy to make changes without touching the original tarball.

It is acceptable to change the original tarball if one or more of the following hold true:

- It contains non-free parts that cannot be redistributed. Remove those parts, and note it in the packaging. Often such packages use "dfsg" (which stands for Debian Free Software Guidelines) in the package name and/or versioning to indicate that non-free parts have been removed.
- The authors only provide bzip2'ed source.
  - Just bunzip2 the .tar.bz2 and gzip -9 the resulting tar.
  - The md5sums of the .tar you provide and the original .tar must match!
  - Eventually provide a get-orig-source rule in debian/rules that does this conversion automatically.
- Directly imported from SVN
  - Provide get-orig-source in debian/rules.

The following are not reasons to change the original tarball:

- Wrong Directory Layout

dpkg-source is quite flexible and manages to produce the correct directory layout even if:

- The directory inside the tarball is not named <upstream>-<version>.
- There is no subdirectory inside the tarball.

- Files need to be removed to keep the .diff.gz small (e.g., files created by autotools). Everything that needs to be deleted should be removed in the clean rule. Since the .diff.gz is created with diff -u, you will not see removed files in the .diff.gz.
- Files need to be modified. Files that need to be modified should to go into .diff.gz. That is its purpose!
- Wrong permissions on files. You can use `debian/rules` to do this.

What do I do with an .orig.tar.gz that already includes a `debian/` dir?

Do not repackage it. You can ask the author(s) to delete the debian/ dir and provide a diff.gz instead. This makes it easier to review their work, and it separates packaging from program source.

It is always a good idea to contact the program's author(s) and ask if you may correct autoconf issues, directory layout, an outdated Free Software Foundation address in COPYRIGHT files, or other things that are not specific to the packaging but would be convenient for you so you do not need to "patch" the source in .diff.gz.

4.3. Copyright Information

The `debian/copyright` file should contain:

- The licensing information for *all* files in the source. Sometimes author(s) put a license in `COPYING` but have different licensing information for some files in the source.
- The copyright holder(s) and year(s).
- The *entire* license unless it is one of the licenses found in `/usr/share/common-licenses`, in which case you should just include the preamble.

Chapter 4. Patch Systems

Quite often it turns out that the upstream source needs to be patched, either to adjust the program to work with Ubuntu or to fix bugs in the source before they are fixed upstream. But how should we reperesent these changes? We could simply make the changes in the unpacked source package, in which case the patch would be expressed in the `.diff.gz` file. However, this is not ideal. If there is more than one patch you loose the ability to seperate the patches as you just see one big diff that also contains the packaging files (in `debian/`. This can make it more difficult when you want to send the patches upstream. It is also very convenient to seperate the author's source from the changes made for Ubuntu. The best place to put this information is in the `debian/` that is already used for the packaging files. For the rest of this chapter we will be looking at the various ways to set up patches in this way.

1. Patching Without a Patch System

As was mentioned above, one can patch the original source by simply making the changes in the unpacked source directory. A real-life example of this is cron. If you grab cron's source package and look at the `.diff.gz` you will see that several of the original source's files were changed.

```
apt-get source cron
zgrep +++ cron*.diff.gz
```

But as we mentioned before this is not really the best way to represent patches. One better way is to create individual patch files, but them in `debian/patches/` and apply the patches (using patch) in `debian/rules`. This is what is done for udev:

```
apt-get source udev
zgrep +++ udev*.diff.gz
ls udev*/debian/patches/
less udev*/debian/rules
```

The `rules` file has the following rules for applying and unapplying the patches:

```
# Apply patches to the package
patch: patch-stamp
patch-stamp:
        dh_testdir
        @patches=debian/patches/*.patch; for patch in $$patches; do \
                test -f $$patch || continue; \
                echo "Applying $$patch"; \
                patch -stuN -p1 < $$patch || exit 1; \
        done
        touch $@

# Remove patches from the package
unpatch:
        dh_testdir
        @if test -f patch-stamp; then \
                patches=debian/patches/*.patch; \
                for patch in $$patches; do \
                        reversepatches="$$patch $$reversepatches"; \
                done; \
                for patch in $$reversepatches; do \
                        test -f $$patch || continue; \
                        echo "Reversing $$patch"; \
                        patch -suRf -p1 < $$patch || exit 1; \
                done; \
                rm -f patch-stamp; \
        fi
```

That is all very nice, but how do we create new patches for udev using this scheme? The general approach is:

1. copy the clean source tree to a temporary directory
2. apply all patches up to the one you want to edit; if you want to create a new patch, apply all existing ones (this is necessary since in general patches depend on previous patches)

   if you want, you can use debian/rules for this: remove the patches that come *after* the one you want to edit, and call 'debian/rules patch'. The actual name for the patch target varies, I have seen the following ones so far: patch setup apply-patches unpack patch-stamp. You have to look in `debian/rules` how it is called.
3. copy the whole source tree again:

   ```
   cp -a /tmp/old /tmp/new
   ```
4. go into /tmp/new, do your modifications
5. go back into your original source tree, generate the patch with:

   ```
   diff -Nurp /tmp/old /tmp/new > mypatchname.patch
   ```

1.1. Example 1.

Let us make a new patch for udev called `90_penguins.patch` which replaces *Linux* with *Penguin* in the udev `README` file:

```
cd udev*/
cp -a . /tmp/old
pushd /tmp/old
debian/rules patch
cp -a . /tmp/new; cd ../new
sed -i 's/Linux/Penguin/g' README
cd ..
diff -Nurp old new > 90_penguins.patch
popd
mv /tmp/90_penguins.patch debian/patches
rm -rf /tmp/old /tmp/new
```

1.2. Example 2.

What happens if we want to edit an existing patch? We can us a similar procedure as Example 1 but we will apply the patch to be edited first:

```
cp -a . /tmp/old
pushd /tmp/old
cp -a . /tmp/new; cd ../new
patch -p1 < debian/patches/10-selinux-include-udev-h.patch
sed -i '1 s/$/***** HELLO WORLD ****/' udev_selinux.c
cd ..
diff -Nurp old new > 10-selinux-include-udev-h.patch
popd
mv /tmp/10-selinux-include-udev-h.patch debian/patches
```

```
rm -rf /tmp/old /tmp/new
```

So this way of patching the source, while technically fine, can become very complicated and unmanageable. To make patching easier and more straightforward patch systems were developed. We will take a look at couple popular ones.

2. CDBS with Simple Patchsys

The CDBS build helper system (see *Section 3, “Packaging With CDBS” [p. 32]*) has a very simple patch system built in. You simply need to add an include for *simple-patchsys.mk* in `debian/rules`. An example is pmount. Its entire `rules` looks like:

```
#!/usr/bin/make -f
include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/class/autotools.mk
include /usr/share/cdbs/1/rules/simple-patchsys.mk

common-post-build-arch::
        # Generate a POT file
        cd po; intltool-update -p --verbose
```

Simple patchsys also has a patch editor built in called cdbs-edit-patch. You can give cdbs-edit-patch either the name of an existing patch to edit or a new patch to create. It will apply the existing patch, if it exists, and put you in a new shell. You can then make any changes you want added to the patch and finally type *Ctrl-D* to exit the shell and create the new patch. The patches are stored in `debian/patches/`

3. dpatch

A popular patch system is dpatch. It has a dpatch-edit-patch script like cdbs has but stores the patches a little differently. It uses a file named `debian/patches/00list` to find the name and order of patches to apply. This means you can order your patches in whichever way you want and can disable a patch without removing it altogether. However, it also mean you need to update `00list` if you add a patch. If dpatch-edit-patch is called with two arguments it will edit/create the the patch named by the first argument relative to the patch named by the second argument. In other words:

```
dpatch-edit-patch new.dpatch old.dpatch
```

will apply patches up to `old.dpatch` and then create `new.dpatch`. Note that dpatch patches usually have a .dpatch suffix. This is because dpatch stores the patches in a slightly different format then a normal patch that adds a special header.

A real-life example of dpatch usage is the xterm package.

4. Patching other people's packages

The most important thing to keep in mind when patching packages maintained by other people is to keep the patch system (or lack thereof) that the maintainer has set up. This will ensure consistency and make the package maintainer more likely to accept your patch.

It is also a good idea to separate patches logically rather than creating one giant patch. If the upstream authors apply one of your changes but not another it is much easier to just drop a patch then edit a monolithic patch to update it.

Chapter 5. Updating Packages

If you have been around Linux distributions for any amount of time, you have realized that there are sometimes bugs in programs. In the Debian and Ubuntu distributions, bugs are often fixed through the packaging by patching the source code. Sometimes there are bugs in the packaging itself that can cause difficulties.

To patch the program's source code, you could simply download the current Ubuntu source package (with apt-get source) and make the needed changes. You can then add a new entry to the `debian/changelog` using dch -i or dch -v <version>-<revision> to specify the new revision. When you run debuild -S from the source directory you will have a new source package with a new `.diff.gz` in the parent directory that contains your changes. A problem with this approach is that the distinction between source and patches is unclear.

A solution to this problem is to separate the changes to the source code into individual patches stored in the `debian` directory. One such patch system is called dpatch. The patches are stored in the `debian/patches/` directory and have a special format.

To create a dpatch, perform the following steps sequentially.

Create a temporary work space and two copies of the current source directory:

```
mkdir tmp
cd tmp
cp -a ../<package>-<version> .
cp -a <package>-<version> <package>-<version>.orig
```

Make the changes in the `<package>-<version>` directory.

Create a patch file using diff and place it in the `debian/patches` directory:

```
diff -Nru  <package>-<version>.orig <package>-<version> > patch-file
```

Create the dpatch using dpatch patch-template and a file named `00list` that lists the dpatches:

```
dpatch patch-template -p "01_patchname" "patch-file description" \
 < patch-file > 01_patchname.dpatch
echo 01_patchname.dpatch >00list
```

You can now place `01_patchname.dpatch` and `00list` in the `debian/patches` directory of your source package:

```
mkdir ../<package>-<version>/debian/patches
cp 01_patchname.dpatch 00list ../<package>-<version>/debian/patches
cd ..
```

```
rm -rf tmp
```

You can also edit a pre-existing patch using dpatch-edit-patch.

Once all the changes have been made, a changelog entry added, and dpatch added to the `debian/control` file (if needed), then you can rebuild the source package with debuild -S.

To get your fixed source package uploaded to the Ubuntu repositories, you will need to get your source package sponsored by a person who has upload rights. See *Section 1, "Uploading and Review" [p. 4]* for more details. Sometimes, rather than giving the entire source package (.diff.gz, .dsc, and .orig.tar.gz), it is easier and more efficient to just give the difference between the source package that is currently in the repositories and your fixed source package. A tool has been created to do just that called debdiff. Using debdiff is similar to using diff but is made specifically for packaging. You can debdiff the source package by:

```
debdiff <oldpackage>.dsc <newpackage>.dsc > package.debdiff
```

or the binary package by:

```
debdiff <oldpackage>.deb <newpackage>.deb > package.debdiff
```

Debdiffs are great to attach to bug reports and have ready for a sponsor to upload.

Chapter 6. Ubuntu Packaging

1. Uploading and Review

Once you have created a source package (either a completely new package or just an update/bugfix), you will want to distribute your package so that other people can enjoy your hard work. The most effective way to do that for Ubuntu is to contribute your package to the Universe repository. The community developers who are responsible for the Universe repository are known as Masters of the Universe (*MOTU* [https://wiki.ubuntu.com/MOTU]). *REVU* [http://revu.tauware.de] is a web-based tool that gives people a place to upload their source packages for other people to look at and for MOTUs to review in a structured manner.

1.1. Contributing as an Uploader

First, you will need to have your *GPG Key* [https://wiki.ubuntu.com/GPGKey] added to the REVU keyring. This step ensures that your packages really come from you and helps track uploads.

REVU uses your *Launchpad* [https://launchpad.net] account to look up your gpg key so make sure you have a Launchpad account and you have put your gpg key in your user profile there. Once this is one you can join the *Universe Contributors* [https://launchpad.net/people/ubuntu-universe-contributors] team and then email *admin@tiber.tauware.de* [mailto:admin@tiber.tauware.de] asking for your gpg key to be updated on REVU. When this is done you will be able to upload your packages. You don't#need a password to upload packages, only to log in to the website and to reply to comments.

It is not necessary to have your key signed by others to upload to REVU, but it is a good idea to have it done at some point.

1.1.1. Uploading your packages

Uploading to REVU uses dput. The Ubuntu version of dput already knows about REVU so you do not need to change any configuration files. Only upload signed packages, and unlike other repositories, you should always include the original tarball, otherwise reviewers will not be able to look at your extracted source package. In order to do so, use the options "-S -sa" with debuild or dpkg-buildpackage to build only the source package and to include the original source in the upload.

After the source package is built, you can use dput with the above config file changes to upload it by specifying just the `_source.changes` file that was created:

```
dput revu *_source.changes
```

If you are reuploading a changed package (after receiving reviews), you may get an error like this:

```
Upload package to host revu
Already uploaded to tauware.de
Doing nothing for myapp_source.changes
```

To fix, add the -f option to dput to force the upload or remove the `.upload` file that was created by dput.

Processing of uploads is done every five minutes, so if your upload does not show up, please contact the REVU administrators by email, or join the Freenode IRC channel #ubuntu-motu.

1.1.2. How to login to REVU

After your first upload, you are registered automatically in the database and assigned a random password. On the *REVU* [http://revu.tauware.de] website, use the email address you used in the changelog file of your upload as the login and click the 'recover password' link. You will be taken to a page that has your encrypted password with instructions for decrypting it.

1.1.3. View and comment uploads

Packages uploaded to REVU are public. You can browse them without logging in to the system. However, commenting on uploads is available only to registered users. As an uploader, you can only comment on your own uploads. This can be useful to give reviewers some info on the changes you have made between two uploads of your packages.

1.1.4. Additional rules

- You must have reviewed the package for known security vulnerabilities and must provide patches for them.
- The package can be refused on the grounds of known security problems.
- You must have included a copyright and license file, and those must allow inclusion of the package in the Universe component and redistribution via Ubuntu mirrors.
- The package must be known to build on top of the main component of the current Ubuntu stable release. It is permissible to require other packages already in Universe.

1.1.5. Getting Help

If you need assistance on these steps, or if you have questions about REVU, you can ask in #ubuntu-motu on the Freenode IRC network.

2. Merges and Syncs

Requirements: build-essential, automake, gnupg, lintian, fakeroot, patchutils, debhelper and *pbuilder [p. 14]*.

Ubuntu is based on the Debian Linux distribution and uses the same package management system (APT). At the beginning of each Ubuntu development cycle, the packages in Ubuntu are updated to those in the Debian unstable branch. However, because Ubuntu is not the same as Debian, some of the packages need to be modified to work in Ubuntu. There might also be bug fixes that Ubuntu developers have introduced into the packages. You can determine whether this has taken place by noting the package version. If the package version includes ubuntu in it (an example would be gimp-2.2.9-3ubuntu2), then the Ubuntu developers have made changes, and it is no longer the same as the Debian package. There are more than 1000 such modified packages in the Universe repository.

At the start of each Ubuntu development cycle, a decision is made regarding these Ubuntu versioned packages. Of course if the Debian version hasn't changed since the last Ubuntu release, then nothing needs to be changed. However, if there is a newer version of the package in Debian, then one of two things should happen. If all of the reasons for Ubuntu modifications (bug fixes, dependencies, etc.) are fixed in the new Debian package, then we can just take the Debian package directly. This decision is called a *sync*. However, if the new Debian version has the same issues that caused the Ubuntu version to be made, then those changes need to be applied to the new Debian version, too. This decision is called *merging*.

2.1. Merging Tutorial

The merging process involves looking at the changes to both the Debian and Ubuntu source packages and determining what has changed and which changes are Ubuntu-specific. Let us now look at an example, a popular CD creation program called xcdroast.

To start, make a folder to hold our project, then navigate there:

```
mkdir ~/xcdroast
cd ~/xcdroast
```

Now download all of the source packages involved into this directory:

- The xcdroast source tarball that is used by all versions:
  - *xcdroast_0.98+0alpha15.orig.tar.gz* [http://snapshot.debian.net/archive/2006/01/16/debian/pool/main/x/xcdroast/xcdroast_0.98+0alpha15.orig.tar.gz]
- The Ubuntu Breezy source package files:
  - *xcdroast_0.98+0alpha15-1.1ubuntu1.dsc* [http://doc.ubuntu.com/files/packagingguide/xcdroast_0.98+0alpha15-1.1ubuntu1.dsc]

  - *xcdroast_0.98+0alpha15-1.1ubuntu1.diff.gz*
    [http://doc.ubuntu.com/files/packagingguide/xcdroast_0.98+0alpha15-1.1ubuntu1.diff.gz]
- The Debian source package files that the Breezy packages are derived from:
  - *xcdroast_0.98+0alpha15-1.1.diff.gz*
    [http://doc.ubuntu.com/files/packagingguide/xcdroast_0.98+0alpha15-1.1.diff.gz]
  - *xcdroast_0.98+0alpha15-1.1.dsc*
    [http://doc.ubuntu.com/files/packagingguide/xcdroast_0.98+0alpha15-1.1.dsc]
- The new Debian source package files that the Dapper packages will be derived from:
  - *xcdroast_0.98+0alpha15-3.dsc*
    [http://doc.ubuntu.com/files/packagingguide/xcdroast_0.98+0alpha15-3.dsc]
  - *xcdroast_0.98+0alpha15-3.diff.gz*
    [http://doc.ubuntu.com/files/packagingguide/xcdroast_0.98+0alpha15-3.diff.gz]

These steps can also be done by searching for the Debian packages at *packages.debian.org* and the Ubuntu packages at *packages.ubuntu.com*.

A very useful package to have installed when doing merges (or any Ubuntu packaging) is devscripts. If you do not have that already installed, install it before proceeding.

By looking at the Ubuntu changelog you should be able to see which differences to expect between the Ubuntu package and the Debian package from which it was derived. For xcdroast, the Ubuntu changelog can be found at *changelogs.ubuntu.com* [http://changelogs.ubuntu.com/changelogs/pool/universe/x/xcdroast/xcdroast_0.98+0alpha15-1.1ubuntu1/changelog]. It says that a .desktop file was fixed and properly installed to close a bug reported in *Malone* [https://launchpad.net/malone/bugs/2698].

Now inspect the actual changes in the source packages:

```
debdiff xcdroast_0.98+0alpha15-1.1.dsc xcdroast_0.98+0alpha15-1.1ubuntu1.dsc | \
 ubuntu.debdiff | less ubuntu.debdiff
```

The lines that start with - have been removed from the Debian package, and those that start with + have been added to the Ubuntu package.

The following is what we see:

- In debian/rules install is being used instead of cp to install the xcdroast icon. Also, there is a new line installing the .desktop file.
- In debian/changelog the changes made are added to the changelog entry.
- In debian/dirs usr/share/applications has been added for the install lines above to work properly.
- xcdroast.desktop is added

Now we know how the Ubuntu source was changed. Now we need to see what has changed in the Debian sources.

```
debdiff xcdroast_0.98+0alpha15-1.1.dsc xcdroast_0.98+0alpha15-3.dsc > debian.debdiff
less debian.debdiff
```

There is a lot more in this debdiff than in the last one. One way we can get a better idea of what has changed is to see what files were changed in the debdiff:

```
grep diff debian.debdiff
```

This indicates that debian/postinst, debian/rules, debian/changelog, debian/doc-base.manual, debian/control, and debian/menu were changed in the new Debian version.

Thus we can see that we need to check debian/rules to see if the Ubuntu changes were made. We can also see that debian/dirs was not changed from the old Debian version. Let us now look at the files. We can unpack the source package by using dpkg-source:

```
dpkg-source -x xcdroast_0.98+0alpha15-3.dsc
```

This will decompress the xcdroast_0.98+0alpha15.orig.tar.gz file, create a xcdroast-0.98+0alpha15 directory, and apply the changes found in xcdroast_0.98+0alpha15-3.diff.gz.

Now navigate to the debian directory:

```
cd xcdroast-0.98+0alpha15/debian
```

One can see in `rules` that changes made by Ubuntu were not applied to the new Debian version. This means that:

```
cp debian/xcdroast.xpm `pwd`/debian/$(PACKAGE)/usr/share/pixmaps
```

...should be changed to:

```
#cp debian/xcdroast.xpm `pwd`/debian/$(PACKAGE)/usr/share/pixmaps

#install desktop and icon
install -D -m 644 $(CURDIR)/debian/xcdroast.desktop \
 $(CURDIR)/debian/xcdroast/usr/share/applications/xcdroast.desktop
install -D -m 644 $(CURDIR)/debian/xcdroast.xpm \
 $(CURDIR)/debian/xcdroast/usr/share/pixmaps/xcdroast.xpm
```

Now in `dirs`, the following line needs to be added for the .desktop file to be installed:

```
usr/share/applications
```

Now we need the actual .desktop file (saved as *debian/xcdroast.desktop*). From the ubuntu.debdiff (or the Ubuntu source package), we see that it is:

```
[Desktop Entry]
```

```
Encoding=UTF-8
Name=X-CD-Roast
Comment=Create a CD
Exec=xcdroast
Icon=xcdroast.xpm
Type=Application
Categories=Application;AudioVideo;
```

The last change that needs to be made is in `changelog`. Not only do we need to add what we have just done (merge with Debian), but we should also add in the previous Ubuntu changelog entries. To do this, run dch -i -D dapper and put something to the effect of:

```
xcdroast (0.98+0alpha15-3ubuntu1) dapper; urgency=low

  * Resynchronise with Debian.
```

Make sure to change the version number to the correct Ubuntu version. Also add:

```
xcdroast (0.98+0alpha15-1.1ubuntu1) breezy; urgency=low

  * Fix and install existing .desktop file. (Closes Malone #2698)

  -- Captain Packager <packager@coolness.com>  Sat,  1 Oct 2005 19:39:04 -0400
```

between the 0.98+0alpha15-1.1 and 0.98+0alpha15-2 log entries.

Now you can build and test the new source packages. There are different ways to do this, but one example is:

```
cd ..
debuild -S
cd ..
sudo pbuilder build xcdroast_0.98+0alpha15-3ubuntu1.dsc
```

This will recreate the source package, sign it with your default GPG key, and build the package in a pbuilder environment to make sure it builds correctly. Make sure to always test your packages before submitting patches. The last step is to make a debdiff that can be attached to an existing bug report or given to the MOTUs in the #ubuntu-motu IRC channel. To do this, we get the difference between the Debian unstable source package and the new Ubuntu version:

```
debdiff xcdroast_0.98+0alpha15-3.dsc xcdroast_0.98+0alpha15-3ubuntu1.dsc > \
 xcdroast_0.98+0alpha15-3ubuntu1.debdiff
```

3. Packaging for Kubuntu

As one might imagine, the main packaging issues specific to Kubuntu are with KDE and Qt.

3.1. Build Dependencies

Kubuntu programs are mostly KDE ones. Therefore, they need to Build-Depend on `kdelibs4-dev`. Since KDE's focus is to have programs interacting, some programs might also need to Build-Depend on other parts of KDE, such as `kdepim-dev`. Be sure to get the list of necessary dependencies for your program.

3.2. Desktop Files

KDE has some specific paths. Most settings for KDE are installed in either `/etc/kde3/` or `/usr/share/apps/`. It is important to note that the general desktop files for KDE should go to `/usr/share/applications/kde/`. The install path for the desktop files should be fixed if they do not use this (except for desktop files like service menus).

KDE desktop files also need specific entries to fit in the KMenu. A minimal desktop file for a KDE program could be something like this:

```
[Desktop Entry]
Encoding=UTF-8
Name=Kfoo
Name[xx]=Kfoo
GenericName=Bar description
Exec=kfoo
Icon=kfoo
Terminal=false
Categories=Qt;KDE;Utility;
```

Note that the Categories field must begin with Qt;KDE;. There are specific desktop file entries for KDE programs and modules that allow su to declare the given programs as KCModules or autostart them when logging in.

3.3. Generating .pot Files

The Ubuntu translation website, *Rosetta* [https://launchpad.net/rosetta/], now supports KDE, which means KDE packages need to support Rosetta by generating .pot template files for translators. If you use cdbs in edgy, your package should now automatically build and check for a .pot file in po/ directory.

You will need the *kdepot patch* [../files/kubuntu_01_kdepot.diff] (or similar; it may not apply cleanly depending on the age of the admin directory).

If your package uses debhelper or cdbs and includes its own kde.mk file, you need to add the rules yourself.

For cdbs, add these lines to debian/rules:

```
common-post-build-arch::
 mkdir -p po
        XGETTEXT=/usr/bin/kde-xgettext sh admin/cvs.sh extract-messages

clean::
        rm -rf po
```

For debhelper, add the following to the end of the *install* rule:

```
mkdir -p po
XGETTEXT=/usr/bin/kde-xgettext sh admin/cvs.sh extract-messages
```

Also for debhelper, add the following to the *clean* rule:

```
rm -f po/*.pot
```

Chapter 7. Bugs

One thing that you will almost certainly face as a packager is a bug in the software itself or in your packaging. Packaging bugs are often fairly easy and straightforward to fix. However, as packagers often act as the initial contact for software bugs for the users of their distribution(s), they also implement temporary fixes and are responsible for forwarding bug reports and fixes to the original (upstream) authors.

The *Ubuntu Bug Squad* [https://wiki.ubuntu.com/BugSquad] is the Quality Assurance (QA) team for Ubuntu. The people in the team work tirelessly to make Ubuntu a better place. They keep track of all the bugs in the Ubuntu Distribution and make sure that major bugs don't go unnoticed by the developers. Anyone can join the Bug Squad and it is a great entry point for people wanting to contribute to Ubuntu. The Bug Squad can be found on the *#ubuntu-bugs* IRC channel on *irc.ubuntu.com*

1. Bug Tracking Systems

In order to track bugs (both software and packaging), many distributions have developed bug tracking systems to manage bug reports and to notify the package maintainers and reporters of changes. The table below shows some of the Debian and Ubuntu tools for tracking bugs.

Bug Tracking Systems (BTS)

Debian : *http://bugs.debian.org*

Ubuntu : *http://launchpad.net/malone/distros/ubuntu*

Bugs for Specific Packages

Debian : http://bugs.debian.org/<*packagename*>

Ubuntu : use search at Ubuntu BTS

Bugs for Source Packages

Debian : http://bugs.debian.org/src:<*packagename*>

Ubuntu : https://launchpad.net/distros/ubuntu/+source/<*packagename*>/+bugs

Package Information

Debian : *http://packages.debian.org* or http://packages.qa.debian.org/<*packagename*>

Ubuntu : *http://packages.ubuntu.com* or
https://launchpad.net/distros/ubuntu/+source/<*packagename*> for source packages

2. Bug Tips

2.1. Proper source package

Assigning bugs to packages helps direct bug reports to the developer(s) most likely to be able to help. By ensuring that this information is accurate, you increase the chances of the bug being fixed promptly. Often, it is unclear which package contains the bug, and in these cases it is appropriate to file the bug in Ubuntu. If a bug is assigned to a package which is clearly not correct, and you don't know the correct package, change it to Ubuntu.

The correct package for bugs in the Linux kernel is linux, regardless of which particular package is in use (there are many packages which contain Linux kernels).

2.2. Confirming problems

If a bug is marked as Unconfirmed, it is helpful for you to try to reproduce the problem and record the results in Malone. If you are able to confirm the problem, you may change the status to Confirmed. If you are unable to confirm the problem, that is also useful information that should be recorded in a comment.

Forwarding bugs upstream

You can forward bugs to the authors of the software (upstream), if

- you made sure that the bug doesn't occur because of Ubuntu related changes
- the change is too hard to be fixed by yourself or anyone else on the team

If you do this, be sure to include all the necessary information, such as

- how to reproduce the bug
- which version is used (which version of dependent libraries, if the bug indicates problems there)
- who reported it
- where the whole conversation can be found

Make sure to also create a bug watch in Malone for this bug.

2.3. How to Deal with Feature Requests

If you feel that the bug reported is a feature request disguised as a bug report, please introduce the reporter gently to the *specification process* we have. Be sure to mention the following specification resources: FeatureSpecifications, SpecSpec, SpecTemplate, and *http://launchpad.net/specs*

2.4. How to deal with Support Requests

If you feel that the bug reported is a support request disguised as a bug report, please introduce the reporter gently to the Support Tracker we have. Be sure to mention *http://launchpad.net/support*.

2.5. How to deal with suggestions for changing defaults

If you feel that the bug reported is a suggestion for changing defaults disguised as a bug report, please kindly reroute the discussion to an appropriate mailing list or discussion forum. If this change has already been discussed and rejected, explain the reasons to the user and direct him or her to the relevant discussion for further suggestions/comments.

2.6. Finding Duplicates

Finding duplicates of bugs is a very valuable contribution in the Bug community. Users sometimes don't know how to check if the same bug has already been filed, and sometimes they don't care. Weeding out simple ME TOO messages and aggregating information is crucial to the process of fixing a bug.

There are quite a few measures you can take to assist with this aspect. One is to search for bugs filed for the same component. Also try to rephrase your search, and concentrate on actions and words that describe the items involved to reproduce the bug.

Examples:

- Easy ones: *DAAP support* [https://launchpad.net/malone/bugs/24932] is a duplicate of *please enable daap* [https://launchpad.net/malone/bugs/24860].
- More difficult ones: *plug:spdif on emu10k1 gone after breezy upgrade* [https://launchpad.net/malone/bugs/24011] is a duplicate of *Muted sound after dist-upgrade from Hoary to Breezy* [https://launchpad.net/malone/bugs/21804].

If you can't find it in the list of open bugs, you could try to find it in the list of closed ones. Don't feel discouraged if you don't find duplicates quickly in the beginning. After some time, you will recognize the usual suspects and will be able to identify them more easily.

If you encounter a bug that has a terrible/unintelligible title, rephrase it so people find it more quickly.

2.7. #Reminder of the Code of Conduct

Note that the Code of Conduct applies to conversations in bug reports too. If you observe people being disrespectful, please direct them to the *Ubuntu Code of Conduct* [http://www.ubuntu.com/community/conduct].

2.8. Managing Status

As a bug triager or developer bug status an important tool to categorize bugs and have a good overview of the state of packages and software.

Here's a brief list and explanation of the various statuses:

- **Unconfirmed:** Bugs start with this status. Bugs marked Unconfirmed sometimes lack information, are not ready, or are not confirmed yet. Most of them have not yet been triaged.
- **Needs Info:** If you have to ask the reporter questions, please set this bug to "Needs Info". A regular task for Needs Info bugs is to ask back. If there are no answers after a reasonable period, close them saying "If you have more information on this bug, please reopen."
- **Rejected:** Bugs marked as Rejected are closed. Be sure to triple-check a bug before you reject it.
- **Confirmed:** Confirmed bugs require somebody else to confirm. Please don't confirm your own bugs.
- **In Progress:** If you start working on a bug, set it to In Progress so people know someone is working on the bug.
- **Fix Committed:** For upstream projects this means the fix is in CVS/SVN/bzr or committed somewhere. For package maintainers it means that the changes are pending and to be uploaded soon (it is what PENDINGUPLOAD is in Bugzilla)
- **Fix Released:** For upstream projects this means that a release tarball was announced and is publicly available. For package maintainers this means that a fix was uploaded. Please don't be hesitant to add a changelog as a comment, so people know which changes affect their bug(s).

2.9. Managing Importance

Launchpad uses the following guidelines for assigning importance:

- **Untriaged:** the bug report has not be triaged yet. This is the default importance for new bugs.
- **Wishlist:** a request to add a new feature to one of the programs in Ubuntu. Use this for bugs which aren't really bugs but ideas for new features which do not yet exist.
- **Low:** bugs that affect functionality, but to a lesser extent than most bugs
- **Mediam:** a functionality bug of the standard variety. Most bugs are of "Medium" severity.
- **High:** a bug that has a severe impact on a small portion of Ubuntu users (estimated) or has a moderate impact on a large portion of Ubuntu users (estimated)
- **Critical:** a bug which has a severe impact on a large portion of Ubuntu users

Appendix A. Appendix

1. Additional Resources

Debian Resources

- *Debian New Maintainers Guide* [http://www.debian.org/doc/manuals/maint-guide/] - Good resource for learning to package.
- *Debian Policy* [http://www.debian.org/doc/debian-policy/] - The essential Policy manual for Debian and Debian-based distros.
- *Debian Developer's Reference* [http://www.debian.org/doc/manuals/developers-reference/] - Specific information for Debian Developers but has some items of interest for packagers.
- *Library Packaging Guide* [http://www.netfort.gr.jp/~dancer/column/libpkg-guide/libpkg-guide.html] - Guide for packaging libraries.
- *Debian Women Packaging Tutorial* [http://women.alioth.debian.org/wiki/index.php/English/PackagingTutorial] - Another good introduction to Debian packaging.

Other Resources

- *IBM Packaging Tutorial* [http://www-106.ibm.com/developerworks/linux/library/l-debpkg.html]
- *Duckcorp CDBS Documentation* [https://perso.duckcorp.org/duck/cdbs-doc/cdbs-doc.xhtml]
- *Ubuntu MOTU Documentation* [https://wiki.ubuntu.com/MOTU/Documentation]
- *Kubuntu Packaging Guide* [https://wiki.ubuntu.com/KubuntuPackagingGuide]

2. Chroot Environment

A chroot environment is commonly used for development-related work and is basically an install of build-related software. It is always a good idea to do development work in a chroot environment, as it often requires the installation of development packages (whose main purpose is for building packages). An example is when a certain application requires the headers and development version of a library to build (e.g. libabc-dev). A normal user would not require the development version of libabc. Thus it is better to install such development packages in a chroot, leaving the normal operating environment clean and uncluttered. First, install the required packages:

```
sudo apt-get install dchroot debootstrap
```

Make sure to install at least the version of debootstrap that is from the Ubuntu release for which you are trying to create the chroot. You may have to download it from *packages.ubuntu.com* [http://packages.ubuntu.com] and manually install it with dpkg -i.

The next steps are to create, configure, and enter the chroot environment.

```
sudo mkdir /var/chroot
echo "mychroot /var/chroot" | sudo tee -a /etc/dchroot.conf
sudo debootstrap --variant=buildd edgy /var/chroot/ http://archive.ubuntu.com/ubuntu/
```

Creating a chroot environment will take some time as debootstrap downloads and configures a minimal Ubuntu installation.

```
sudo cp /etc/resolv.conf /var/chroot/etc/resolv.conf
sudo cp /etc/apt/sources.list /var/chroot/etc/apt/
sudo chroot /var/chroot/
```

In order to be able to use apt in the chroot, add Ubuntu sources to the chroot's apt sources. For the moment, ignore any warnings about package authentication:

```
echo "deb     http://archive.ubuntu.com/ubuntu edgy main restricted \
 universe multiverse" > /etc/apt/sources.list
echo "deb-src http://archive.ubuntu.com/ubuntu edgy main restricted \
 universe multiverse" >> /etc/apt/sources.list
apt-get update
apt-get install build-essential dh-make automake pbuilder gnupg lintian \
 wget debconf devscripts gnupg sudo
apt-get update
exit
```

Run the following command to configure locales:

```
sudo chroot /var/chroot/
apt-get install dialog language-pack-en
exit
```

If you want support for a language other than English replace *en* in language-pack-en with the appropriate language code.

Next, fix the user and root passwords for the chroot environment. The last line below is to avoid sudo warnings when resolving in the chroot environment:

```
sudo cp /etc/passwd /var/chroot/etc/
sudo sed 's/\([^:]*\):[^:]*:/\1:*:/' /etc/shadow | sudo tee /var/chroot/etc/shadow
sudo cp /etc/group /var/chroot/etc/
sudo cp /etc/hosts /var/chroot/etc/
```

To enable sudo, set up your root password and the first sudo user in the admin group (for the chroot environment). In the following commands, substitute "<user>" with the username that will be used in the chroot environment:

```
sudo cp /etc/sudoers /var/chroot/etc/
sudo chroot /var/chroot/
dpkg-reconfigure passwd
passwd <user>
exit
```

The system fstab needs to be modified so that the chroot environment will have access to the system home directories, temp directory, etc. Note that the actual system home directory is used in the chroot environment.

```
sudo editor /etc/fstab
```

Add these lines:

```
/home           /var/chroot/home        none    bind            0       0
/tmp            /var/chroot/tmp         none    bind            0       0
proc-chroot     /var/chroot/proc        proc    defaults        0       0
devpts-chroot   /var/chroot/dev/pts     devpts  defaults        0       0
```

Mount the new fstab entries

```
sudo mount -a
```

The default bash profile includes chroot information in the prompt. To make this visible:

```
sudo chroot /var/chroot/
echo mychroot > /etc/debian_chroot
exit
```

Now use your chroot (you may omit the -c mychroot if there's only one or you just want the first one in `/etc/dchroot.conf`). The -d parameter means that your environment will be preserved. This parameter is generally useful if you want chrooted applications to seamlessly use your X server, your session manager, etc.

```
dchroot -c mychroot -d
```

3. dh_make example files

Readme.Debian
: This file is used to document changes that you have made to the original upstream source that other people might need to know or information specific to Debian or Ubuntu.

conffiles.ex
: If the package installs a configuration file, when the package is upgraded dpkg can prompt a user whether to keep his or her version if modified or install the new version. Such configuration files should be listed in `conffiles` (one per line). Do not list configuration files that are only modified by the package or have to be set up by the user to work.

cron.d.ex
: If your package requires regularly scheduled tasks to operate properly, you can use this file to configure it. If you use this file, rename it to `cron.d`.

dirs
: This file specifies the directories that are needed but the normal installation procedure (make installapplication) somehow doesn't create.

docs
: This file specifies the filenames of documentation files that dh_installdocs will install into the temporary directory.

emacsen-*.ex
: This file specifies Emacs files that will be bytecompiled at install time. They are installed into the temporary directory by dh_installemacsen.

init.d.ex
: If your package is a daemon that needs to be run at system startup rename this file to `init.d` and adjust it to your needs.

manpage.1.ex and manpage.sgml.ex
: These files are templates for man pages if the package does not already have one.

menu.ex
: This file is used to add your package to the Debian menu. Ubuntu does not use Debian menu files but uses the *freedesktop.org* [http://www.freedesktop.org] standard *.desktop* [http://standards.freedesktop.org/desktop-entry-spec/latest/] files.

watch.ex
: The package maintainer can use the uscan program and a `watch` file to check for a new upstream source tarball.

ex.package.doc-base
: This file is used to register your package's documentation (other than man and info pages) with doc-base.

postinst.ex, preinst.ex, postrm.ex, and prerm.ex
: These maintainer scripts are run by dpkg when the package is installed, upgraded, or removed.

For more details refer to the *Debian New Maintainer's Guide* [http://www.debian.org/doc/maint-guide/ch-dother.en.html].

4. List of debhelper scripts

- dh_builddeb
- dh_clean
- dh_compress
- dh_desktop
- dh_fixperms
- dh_gconf
- dh_gencontrol
- dh_iconcache
- dh_install
- dh_installcatalogs
- dh_installchangelogs
- dh_installcron
- dh_installdeb
- dh_installdebconf
- dh_installdefoma
- dh_installdirs
- dh_installdocs
- dh_installemacsen
- dh_installexamples
- dh_installinfo
- dh_installinit
- dh_installlogcheck
- dh_installlogrotate
- dh_installman
- dh_installmenu
- dh_installmime
- dh_installmodules
- dh_installpam
- dh_installppp
- dh_installtexfonts
- dh_installwm
- dh_installxfonts
- dh_installxmlcatalogs
- dh_link

- dh_listpackages
- dh_makeshlibs
- dh_md5sums
- dh_perl
- dh_python
- dh_scrollkeeper
- dh_shlibdeps
- dh_strip
- dh_testdir
- dh_testroot
- dh_usrlocal

Appendix B. GNU General Public License

Version 2, June 1991

Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor,
Boston, MA 02110-1301
USA

Version 2, June 1991

1. Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software - to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps:

1. copyright the software, and
2. offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

2. TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

2.1. Section 0

This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The “Program”, below, refers to any such program or work, and a “work based on the Program” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification”.) Each licensee is addressed as “you”.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2.2. Section 1

You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2.3. Section 2

You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of *Section 1* above, provided that you also meet all of these conditions:

a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these

conditions, and telling the user how to view a copy of this License. (Exception: If the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

2.4. Section 3

You may copy and distribute the Program (or a work based on it, under *Section 2* in object code or executable form under the terms of *Sections 1* and *2* above provided that you also do one of the following:

a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

2.5. Section 4

You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

2.6. Section 5

You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

2.7. Section 6

Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

2.8. Section 7

If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

2.9. Section 8

If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

2.10. Section 9

The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

2.11. Section 10

If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

2.12. NO WARRANTY Section 11

BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND,

EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

2.13. Section 12

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

3. How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type “show w”. This is free software, and you are welcome to redistribute it under certain conditions; type “show c” for details.

The hypothetical commands “show w” and “show c” should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than “show w” and “show c”; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program “Gnomovision” (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.