

Debian for Developers Tutorial

Wichert Akkerman

wichert@linux.com

Robert van der Meulen

rvdm@wiretrip.org

Copyright © 2001 by Wichert Akkerman

Copyright © 2001 by Robert van der Meulen

Table of Contents

<u>Preface</u>	1
<u>Chapter 1. Packages</u>	2
<u>1.1. The package format</u>	2
<u>1.1.1. Overview</u>	2
<u>1.1.2. The package metadata</u>	2
<u>1.1.3. The package data</u>	3
<u>1.2. Package relations</u>	3
<u>1.3. The package installation process</u>	3
<u>1.4. The package removal process</u>	4
<u>Chapter 2. Building packages</u>	5
<u>2.1. The debian/ directory</u>	5
<u>2.1.1. The debian/control file</u>	5
<u>2.1.2. The debian/rules script</u>	6
<u>2.1.3. The debian/shlibs file</u>	6
<u>2.2. The debian/changelog file</u>	6
<u>2.3. Version numbers</u>	7
<u>2.4. Questions</u>	7
<u>Chapter 3. Interface with the rest of the OS</u>	8
<u>3.1. Users and groups</u>	8
<u>3.2. Filesystem Hierarchy Standard</u>	8
<u>3.3. cron jobs</u>	9
<u>3.4. Questions</u>	9
<u>Chapter 4. daemons and init scripts</u>	10
<u>4.1. init scripts</u>	10
<u>4.2. The start–stop–daemon tool</u>	10
<u>4.3. Handling installation and removal</u>	11
<u>4.4. Questions</u>	11
<u>Chapter 5. Cleaning up when removing or purging a package</u>	12
<u>5.1. What data needs to be removed, and what data needs to be kept?</u>	12
<u>5.2. What is the difference between handling a 'remove' and a 'purge'?</u>	12
<u>5.3. Questions</u>	12
<u>Chapter 6. Complex package operations</u>	13
<u>6.1. Virtual packages</u>	13
<u>6.2. Renaming a package</u>	13
<u>6.3. Splitting a package</u>	13
<u>6.4. Example of a complex package: proftpd</u>	13
<u>6.5. Questions</u>	14
<u>Chapter 7. Build Dependencies</u>	15
<u>7.1. Why we need build constraints</u>	15
<u>7.2. build–essentials</u>	15
<u>7.3. architecture–specific constraints</u>	15

Table of Contents

7.4. Questions	15
Chapter 8. DebConf	16
8.1. Why the need for debconf	16
8.2. Using debconf	16
8.3. The templates file	16
8.4. Handling i18n	16
8.5. Questions	17
Chapter 9. Using debhelper	18
9.1. What is debhelper?	18
9.2. Building a package from scratch using dh make	18
9.2.1. dh make	18
9.2.2. dh install*	18
9.2.3. Other helper utilities	19
Chapter 10. Package archives	20
10.1. Archive structure	20
10.2. Using dpkg-scanpackages, and the override file	21
Chapter 11. Maintainership	22
11.1. Being a responsible package maintainer	22
11.2. Red tape	22
Debian for developers bibliography	22

Preface

These are the handouts for the Debian for Developers tutorial from the LinuxWorld Conference and Expo 2001.

Chapter 1. Packages

Packages are the basic units in which software and data is installed using a package manager. This chapter describes the internals of a package.

1.1. The package format

1.1.1. Overview

The Debian package management system uses the `.deb` format, currently at revision 2. The `.deb` format is very simple and based on the following principles:

- A package has to be extractable using traditional UNIX tools
- The format must be easily extensible

The format is a `ar` archive which contains 3 files:

debian-binary

Contains the version number of the package format. Currently this is "2.0".

control.tar.gz

Compressed `tar` archive with the package metadata

data.tar.gz

Compressed `tar` archive with the package data

1.1.2. The package metadata

The package metadata describes the data that is inside the package (in the `data.tar.gz` component). The only obligatory file in the metadata is the `control` file. This file is a text-file in RFC822 style which describes the package in detail.

Example 1-1. Example control file

```
Package: dpkg
Version: 1.9.12
Section: base
Priority: required
Architecture: i386
Essential: yes
Pre-Depends: libc6 (>= 2.2.3-1), libncurses5 (>= 5.2.20010310-1),
libstdc++2.10-
glibc2.2
Conflicts: sysvinit (<< 2.72)
Replaces: dpkg-doc-ja
Installed-Size: 3112
Origin: debian
Maintainer: Wichert Akkerman <wakkerma@debian.org>
Bugs: debbugs://bugs.debian.org
Description: Package maintenance system for Debian
 This package contains the programs which handle the installation and
```

removal of packages on your system.

The primary interface for the dpkg suite is the `dselect` program; a more low-level and less user-friendly interface is available in the form of the `dpkg` command.

In order to unpack and build Debian source packages you will need to install the developers' package `dpkg-dev` as well as this one.

Control can also contain the (optional) maintainer scripts `preinst`, `postinst`, `prerm`, and `postrm`. These are executables that dpkg will run at various stages during package installation and removal.

Other files may also be present. dpkg will not do anything with those itself but will copy them to its database so other tools can use them. This is used for extensions such as shlibs files and debconf.

1.1.3. The package data

The `control.tar.gz` contains the files that will be installed on the filesystem when a package is installed.

1.2. Package relations

Packages are not stand-alone objects, but can work together to form a complete system. There is a rich set of relations between packages for this purpose:

Pre-Depends

Lists other packages that have to be fully installed before dpkg starts installing this package.

Depends

Lists other packages that have to be fully installed before dpkg can configure this package.

Conflicts

Lists other packages that can not be installed at the same time as this package.

Provides

Indicated that this package provides a virtual package.

Replaces

A list of packages whose file a package is allowed to overwrite.

Recommends, Suggests

A list of packages that might also be useful or interesting to people using this package.

1.3. The package installation process

When installing a package the process is simple:

Package installation procedure

1. `<new-preinst> install`
2. Package is unpacked
3. `<new-postinst> configure`

When we are upgrading an already installed package the procedure is slightly more complex:

Package upgrade procedure

1. `<old-prerm> upgrade <new-version>`
2. `<new-preinst> install <old-version>`
3. Package is unpacked
4. `<old-postrm> upgrade <new-version>`
5. `<new-postinst> configure <old-version>`

If any step in this process fails `dpkg` will try to return to previous safe situation. The details on this process are described in the `dpkg` reference manual.

1.4. The package removal process

Package removal is very similar to installation except the process is done in reverse:

Package removal procedure

1. `prerm remove`
2. Package files are removed (except for `conffiles`)
3. `postrm remove`
4. Remove maintainer scripts except for `postrm`

When the package is being purged two extra steps are performed:

Package purge procedure

1. Configuration files (and associated backupfiles) are removed.
 2. `postrm purge`
 3. Package is removed from the `dpkg` database.
-

Chapter 2. Building packages

2.1. The debian/ directory

The `debian/` subdirectory in a sourcetree contains the files that are needed to build a package. Those files have rules for compiling the sources and building the package, information about the packages that will be made and a changelog.

2.1.1. The debian/control file

The `control` describes the source and the packages that are generated from it. Just like other files this file is in RFC822 syntax, with an empty line separating multiple sections. The first section describes the source, and the other sections the packages.

Table 2–1. Fields in the source section of the `control` file

Field	required	Description
Source	yes	Name of the source package
Section	no	Package category for this package, used by frontends.
Priority	no	Indication of package importance. Legal values are extra, important, optional, required and standard.
Maintainer	yes	Name of the source package
Standards-Version	no	Version of Debian policy this package is compliant with
Build-Depends, Build-Conflicts, Build-Depends-Indep, Build-Conflicts-Indep	no	Specify demands on the build environment
Bugs	no	Name of the source package
Origin	no	Name of the source package

Table 2–2. Fields in the package section of the `control` file

Field	required	Description
Package	yes	Name of the package
Architecture	yes	Space separated list of architectures for which package are build from this source
Essential	no	Indicates if a package is essential to the system
Pre-Depends, Depends, Recommends, Suggests, Replaces, Conflicts	no	Specify relations to other packages
Description	yes	Description of the package

Entries in a package-section of the control file may contain variables that will be substituted when the package is build using the `${variable}` syntax. Variables can be set either by tools such as `dpkg-shlibdeps` or

with commandline options for **dpkg-gencontrol**.

2.1.2. The `debian/rules` script

`debian/rules` is an executable that builds the packages from a source tree. It takes an argument that describes what it should do:

build

Perform a (non-interactive) configuration and compilation of the package.

binary

Build all the packages. This might include the build state as well. Generally this will invoke the `binary-arch` and `binary-indep` rules.

binary-arch

Build all architecture-dependent packages.

binary-indep

Build all architecture-dependent packages.

clean

Undo all the affects from the build and binary targets. The result should be a clean tree with just the uncompiled source.

Although not obligatory all Debian packages use an executable makefile which are very well suited for this use.

2.1.3. The `debian/shlibs` file

When creating a package we need to be sure that all the libraries it uses are listed in the `Depends` field. This is automated using the `shlibs` systems. Each library package contains a `shlibs` file in its metadata that lists the libraries it contains and the proper dependency that is needed for packages using those libraries.

Example 2-1. Example `shlibs` file for `libldap2`

```
liblber 2 libldap2 (>= 2.0.11-1)
libldap 2 libldap2 (>= 2.0.11-1)
libldap_r 2 libldap2 (>= 2.0.11-1)
```

This information is used by the **dpkg-shlibdeps** tool. When it is running during a package build it will check for each library an executable uses which packages provides it and add the associated `shlibs` entry to the list of variable substitutions that **dpkg-gencontrol** uses when generating the control data for a package.

2.2. The `debian/changelog` file

Any changes to the packaging are listed in the `debian/changelog` file. This file is also used to extract the versionnumber for packages being build.

A special format is used that is easily parseable by the packaging tools:

```
<package> (<version>) <distribution(s)>; urgency=<urgency>

* change details
  more change details
* even more change details

-- <maintainer name> <<email address>> <date>
```

The maintainer name listed in the changelog file is extracted when generating packages and used to fill the Changed-By field in a .changes.

It is also possible to close open bugreports in the Debian bug tracking system by putting special tags in the change details. The syntax for that is *Closes: Bug#12345*. This indicates that when the package is installed in the Debian archive bug 12345 should be closed.

2.3. Version numbers

Each package release has a specific version number. A version number consists of two parts: the upstream version, and the debian revision, which are separated by a hyphen (-). Each part may consist of alphanumeric ASCII characters and the characters `.', `+', `-` and `:`. If there is no debian revision then the upstream version is not allowed to have a -.

Comparing version number is done by breaking them into pieces and comparing those from left to right. Pieces are separated by a transition from numbers to characters and by non-alphanumeric characters.

2.4. Questions

1. You are packaging a CVS snapshot from June 29, 2001 from the version 2 development branch of mooscored. What version number would you use?
 2. You want to package the second prerelease for version 2 of mooscored. What version number would you use?
 3. Version 1.3-2 of your package had a bug that requires special attention before upgrading to a later version. How would you handle that?
-

Chapter 3. Interface with the rest of the OS

Often packages need to have an interface to the rest of the system. Depending on the subsystem that a package needs to interface with there are three kinds of interfaces:

- Put a file in a special directory, and possibly calling a tool to process the new (or removed) file.
- Call a tool to (un)register the new package.
- Special interfaces for things like creating system users and groups.

3.1. Users and groups

Users and groups can be added using the **adduser** tool. This tool also makes sure that user or group will be in the right range. Debian defines the following ranges:

0–99

Globally allocated entries which are guaranteed to be the same on all systems.

100–999

Dynamically allocated system accounts. These are used by packages which need accounts but don't need a fixed uid.

1000–29999

Dynamically allocated user accounts.

30000–59999

Reserved range.

60000–64999

Globally allocated static users and groups.

65000–65533

Reserved range.

65534

User nobody.

65535

Not allowed to be used since it is also used as the error return value.

3.2. Filesystem Hierarchy Standard

The FHS is a standard which describes where files and directories should be located on a filesystem. Debian currently uses version 2.1 of the FHS standard.

Since other Linux distributions also adhere to the FHS this also guarantees a level of consistency between distributions.

3.3. cron jobs

If a package needs to perform some activity on a regular basis it can register a cron job. Depending on how often something has to be done there are 2 options:

1. Install an executable in `/etc/cron.daily`, `/etc/cron.weekly` or `/etc/cron.monthly`.
Cron will run all scripts in those directories when needed.
 2. Install a cron entry in `/etc/cron.d` using standard `crontab` syntax.
-

3.4. Questions

1. Where would you put the highscores database for mooscored on the filesystem?
 2. Package quota has to check the quota information on all filesystems before they can be used. When and how would you do this?
 3. Where would you install the **traceroute** tool? (traceroute is a tool to show the network hops between two hosts on an IP network)
-

Chapter 4. daemons and init scripts

Most programs can simply be installed on a computer and do not need any special attention. When a user needs them he will start them himself. Daemons however require some special attention since they need be running at all times in order to provide their services.

4.1. init scripts

If a package needs to perform a specific task during system boot or shutdown it can install an *init-script*. Those scripts are run when booting a system, as well as when changing the runlevel.

Init scripts can be divided in two categories:

1. Bootup scripts which are run early during system initialization.
2. Runlevel scripts which are run when entering or leaving a runlevel.

System boot

1. The kernel boots and starts `/sbin/init`.
2. `init` runs the scripts in `/etc/rcS.d` to perform system initialization. This will bring up the network and mount all filesystems.
3. `init` starts the desired runlevel and runs all the scripts in `/etc/rc#.d/`, where `#` is the runlevel.

The `/etc/rcS.d` and `/etc/rc#.d` directories are filled with symbolic links to `/etc/init.d` where the scripts are stored. The first three characters of the link have a special meaning:

- The first character is either `S` or `K`. This determines if it should be called with *start* or *stop*.
- The next two characters are a number which is used to control the order in which `init` runs the scripts,

The symbolic links are managed using the **update-rc.d** tool. There are three basic ways of using it:

```
update-rc.d  
name remove
```

This removes the symbolic for a script so it is no longer called. Normally **update-rc.d** will only do this after the script is removed, but you can force this with the `-f` option.

```
update-rc.d
```

```
name defaults
```

This will install symlinks for the script named *name* using the defaults (`S20` links for runlevels 2, 3, and 5 and `K20` links for the other runlevels).

```
update-rc.d
```

```
name defaults
```

```
NN-start NN-stop
```

This behaves exactly like the previous command but allows you to specify the ordering number used in the link name for the start and stop links.

4.2. The start-stop-daemon tool

Debian uses the **start-stop-daemon** tool to stop and start daemons. Besides the normal start and stop functionality it has a few useful extensions:

- By using the `--exec` option with `--stop` it will only stop instances of an application that use a specific binary.
 - By using the `--make-pidfile` and `--pidfile` options it can create a pidfile if a daemon can not do that itself.
 - With the `--chroot` option it is possible to start a daemon in a chroot environment.
 - The `--chuid` allow you to run the daemon using a different (non-root) account.
-

4.3. Handling installation and removal

When making a package you have to be careful to make sure the daemon is running after a package is install and it is not running after the package is removed. Special care is needed when handling package upgrades: the running version of the daemon has to be replaced with the new one. This can be handled in two ways:

- Stop the old daemon in a `preinst` script and start the new one in the `postinst` script.
- The daemon running during the upgrade and restart it in the `postinst`.

The second option is potentially more dangerous: since it keeps a daemon running while `dpkg` upgrades the package files it relies on might suddenly be changed or no longer exist. For this reason the first approach is preferred. However some daemons are critical to an environment should have a minimal downtime (DHCP, DNS) and for those the second approach is useful.

4.4. Questions

1. What command would you run to tell **init** to run the `/etc/init.d/mooscore` script to start and stop mooscore?
2. Is this a good way to restart a daemon in your `postinst`?

```
start-stop-daemon --stop --quiet --oknodo --exec /usr/sbin/mooscored
start-stop-daemon --start--quiet --exec /usr/sbin/mooscored
```

3. Is this a good way to restart a daemon in your `postinst`?

```
start-stop-daemon --stop --quiet --oknodo --pidfile /var/run/mooscored.pid
start-stop-daemon --start--quiet \
  --make-pidfile --pidfile /var/run/mooscored.pid \
  --exec /usr/sbin/mooscored
```

4. Is this a good way to restart apache in your `postinst`?

```
apachectl restart
```

Chapter 5. Cleaning up when removing or purging a package

5.1. What data needs to be removed, and what data needs to be kept?

To keep the installation 'clean' from unwanted and unneeded data, a package needs to correctly clean up after itself, after being removed or purged from the system. There are two possible ways to remove a package; one can 'purge' it, and one can 'remove' it. The steps the package management tools go through when removing or purging a Debian package are outlined below.

1. `preRM remove` is called.
2. All files contained in the package are removed, except for the config files.
3. `postRM remove` is called. If the action is 'remove', stop processing here.
4. The configuration files and backup files are removed.
5. `postRM purge` is called.

- All files included in the initial package are automatically removed.
- All configuration files are kept when doing a remove, but deleted when doing a purge.
- Remove log files, debconf database settings, users, groups, things in `/var`.
- Keep: Actual data (databases, documents)

Note: If in doubt, use debconf to ask upon installation. Cleaning up is important, to keep the system from clobbering up with cruft!

5.2. What is the difference between handling a 'remove' and a 'purge'?

- 'purge' is more aggressive, removes config files as well.
 - 'remove' leaves config files in place; upon new installation of the package, they are still there.
-

5.3. Questions

1. what should we do when purging `exim`?
 2. what should we do when purging `mooscored`?
 3. what should we do when purging `cockle`?
-

Chapter 6. Complex package operations

6.1. Virtual packages

A virtual package is a package that appears in the 'Provides:' field of another package. It can be useful to make use of virtual packages when several packages are similar enough to allow a package to depend on any of them. If all these similar packages 'provide' the virtual package, other packages can depend on the virtual package, in stead of every single package.

- A virtual package can be depended on, to depend on any of the packages that provide the virtual package.
- A virtual package can be a real package too; the only way to keep a virtual package from a real package with the same name, is by looking at the version number: the virtual package is not versioned.

A good example of virtual packages in action would be the way Debian handles the presence of a mail transport agent. All mail transport agent packages, such as exim, postfix and sendmail have a 'Provides: mail-transport-agent' line in their control file. Whenever a package has 'mail sending' functionality, it Depends: on 'mail-transport-agent', allowing the user to decide which one is preferred.

6.2. Renaming a package

When renaming a package, always use a 'Replaces:' entry to replace the old one, and a 'Conflicts:' entry to make sure they can not coexist. After the package has been placed in the archive, the package management tools will see there is a package replacing an older one, and – during an upgrade – upgrade to that one automatically.

6.3. Splitting a package

Package splitting is useful to allow a user to install different functional versions of the same program, while keeping the 'Depends:' lines of other packages clean. An existing package that is split like this, should have a 'Replaces:' line for the older version, each part should 'Conflict:' with all other parts that supply the same content, and possibly 'Depend:' on a common package, that is shared by all functional entities.

6.4. Example of a complex package: proftpd

```
Package: proftpd
Section: non-US/main
Architecture: any
Depends: netbase (>= 2.0), ${shlibs:Depends}, debconf, proftpd-common
Conflicts:proftpd-mysql, proftpd-pgsql, proftpd-ldap
Provides: ftp-server
Recommends: proftpd-doc
```

...

```
Package: proftpd-common
Section: non-US/main
Architecture: any
Depends: netbase (>= 2.0),${shlibs:Depends}, debconf, proftpd| proftpd-mysql|proftpd-pgsql|proftpd-ldap
Suggests: proftpd-doc
```

Replaces: proftpd (<= 1.2.1-2)

...

Package: proftpd-mysql

Section: non-US/main

Architecture: any

Depends: netbase (>= 2.0), \${shlibs:Depends}, debconf, proftpd-common

Conflicts: ftp-server, proftpd, proftpd-pgsql, proftpd-ldap

Provides: ftp-server

6.5. Questions

1. Suppose we want to split syslogd into syslogd and klogd. How should we handle relations so that upgrades do not break?
-

Chapter 7. Build Dependencies

7.1. Why we need build constraints

Build daemons need to know what to install, to be able to efficiently build it for other architectures. When a package is being (re)built, the build daemon takes a look at the 'Build-Depends:' line in the debian/control file, and installs the packages that are needed. Build dependencies specify the relation between source packages and binary packages.

7.2. build-essentials

Packages that can be assumed to always be installed during the build of a package are called 'build-essentials'. The 'build-essential' list of packages is defined as the list of packages needed to build a complete package containing a simple 'hello world' program. All 'Essential:' packages are build-essential as well. Currently the list is comprised of:

- gcc
- g++
- make
- dpkg-dev
- libc6-dev

The 'Essential:' packages are not included here. All packages in the list above do not need to be specified in 'Build-Depends:' lines, except when a specific version is needed.

7.3. architecture-specific constraints

Build depends can be limited to specific architectures. If a certain package needs a binary package to be built, but only on a specific architecture, an entry can be added to the Build-Depends: line like this:

```
Build-Depends: foo [ia64]
```

Which lets the package build-depend on package 'foo', but only on architecture 'ia64'. Similarly:

```
Build-Depends: foo [!ia64]
```

This works when you want to build-depend on package 'foo', on all architectures except 'ia64'.

7.4. Questions

1. list build dependencies for our examples
-

Chapter 8. DebConf

8.1. Why the need for debconf

Debconf gives us a mechanism to interact with users; to ask configuration questions, and to store and retrieve the results. Using debconf, a user can select a frontend of choice, and is presented with the questions in a consequent manner.

8.2. Using debconf

Package use debconf by supplying templates for questions that debconf can ask a user as well as a script to drive the interaction. This script (the `config` script) talks with debconf through `stdin` and `stdout`. It is automatically invoked before installing a package if you use `apt`, or by the `postinst`.

8.3. The templates file

Debconf uses templates to describe questions that need asking, and ways to retrieve the result. A simple template could look like this:

```
Template: foo/yes_or_no
Type: boolean
Default: true
Description: Personality poll
  This is just another silly example!
.
```

The possible fields are:

Template

The name of the template.

Type

The variable type for a question. Possible values are `select`, `multiselect`, `string`, `boolean`, `note`, `text` and `password`.

Default

An optional default value.

Description

A description of the variable which is also used as the question show to a user when asking for a value for this variable.

Template

The name of the template.

8.4. Handling i18n

Multilingual support is added trough prefixing Debconf template directives with a specific language identifier (i.e. `Description-nl`):

Description-nl: Persoonlijkheidstest
Dit is stiekum helemaal niet de goede vertaling!

8.5. Questions

1. What Debconf questions will be needed for mooscored and cockle ?
 2. Write the template file that would be needed for mooscored.
-

Chapter 9. Using debhelper

9.1. What is debhelper?

Debhelper is a collection of utilities that are designed to make package building and maintenance easier. All these tools are used in the `debian/rules` file, to automate common package-building tasks. Every Debian package that uses debhelper to aid in building the binary package, needs debhelper in its Build-Depends: line. The command 'dh_make' can set up a `debian/` dir for most packages, building a skeleton packaging structure that can be finished and perfected by hand.

9.2. Building a package from scratch using dh_make

9.2.1. dh_make

Most programs can be easily packaged using debhelper; especially when using `dh_make` to make an initial `debian/` tree. `dh_make` requires the directory containing the source code to be named `<packagename>-<version>`, `<packagenamev` being an all-lowercase string, possibly with digits and dashes. If the directory isn't named like this, it needs to be renamed first.

When `dh_make` is run, it asks a single question about what class the package will be:

single binary

A single binary `.deb` will be built.

multiple binary

Multiple `.deb` packages will be built from a single source.

library

At least two library packages will be built; a development (`-dev`) package, and a library package containing the library itself.

After picking the right package class, an 'original' directory is created in `./<packagename>-<version>.orig/`. `dh_make` proceeds to make the `debian/` directory, and fills it with control files, and some example packaging scripts (these are postfixed with `.ex`). These files can be customized or removed later on. Most of the 'important' debhelper utilities will be placed in the correct positions in the `debian/rules` file, commented out. Editing this file, uncommenting debhelper scripts, can ease the packaging process quite a lot.

9.2.2. dh_install*

The `dh_install*` tools help with installing certain files in the package, or in the package build directories. These are mostly small scripts, ensuring permissions are verified, and installed correctly, directories are created, and programs are registered correctly.

The current list of installation tools is: `dh_installchangelogs`, `dh_installcron`, `dh_installdeb`, `dh_installdebconf`, `dh_installdirs`, `dh_installdocs`, `dh_installemacsen`, `dh_installexamples`, `dh_installinfo`, `dh_installinit`, `dh_installogrotate`, `dh_installman`, `dh_installmanpages`, `dh_installdmenu`, `dh_installdmime`, `dh_installdmodules`, `dh_installdpam`, `dh_installdwm`, `dh_installdxaw` and `dh_installdfonts`.

9.2.3. Other helper utilities

There are quite a few other debhelper utilities, taking care of various packaging–related tasks. A few examples are:

dh_compress

compress files and fix symlinks in package build directories.

dh_fixperms

fix permissions of files in package build directories.

dh_makeshlibs

automatically create shlibs file.

dh_perl

calculates perl scripts & modules dependencies.

dh_strip

strip executables, shared libraries, and some static libraries .

Chapter 10. Package archives

10.1. Archive structure

There are three important files describing the contents of a Debian Archive:

Packages

describes all binary packages in the archive.

Example 10–1. Example Packages file entry

```
Package: dpkg
Essential: yes
Priority: required
Section: base
Installed-Size: 1822
Maintainer: Wichert Akkerman <wakkerma@debian.org>
Architecture: i386
Version: 1.6.15
Replaces: dpkg-doc-ja
Pre-Depends: libc6 (>= 2.1.2), libncurses5, libstdc++2.10
Filename: dists/potato/main/binary-i386/base/dpkg_1.6.15.deb
Size: 714178
MD5sum: 34e83c1b5b0f56a48cleecfdc89ae9d1
Description: Package maintenance system for Debian
 This package contains the programs which handle the installation and
 removal of packages on your system.
.
 The primary interface for the dpkg suite is the `dselect' program;
 a more low-level and less user-friendly interface is available in
 the form of the `dpkg' command.
.
 In order to unpack and build Debian source packages you will need to
 install the developers' package `dpkg-dev' as well as this one.
```

Release

describes the archive itself.

Example 10–2. Example Release file

```
Archive: potato
Component: updates/main
Version: 2.2
Origin: Debian
Label: Debian-security
Architecture: i386
```

Sources

describes all source packages in the archive.

Example 10–3. Example Sources file entry

```
Package: dpkg
Binary: dpkg, dpkg-dev, dpkg-doc
```

```

Version: 1.6.15
Priority: required
Section: base
Maintainer: Wichert Akkerman <wakkerma@debian.org>
Build-Depends: debiandoc-sgml, libncurses-dev
Architecture: any
Standards-Version: 3.1.0
Format: 1.0
Directory: dists/potato/main/source/base
Files:
 3b7c01709cbb59e30a5c9247b6014fe7 571 dpkg_1.6.15.dsc
 a7630586c2c50b27ad8d2800c6ce7d37 1332622 dpkg_1.6.15.tar.gz

```

10.2. Using dpkg-scanpackages, and the override file

dpkg-scanpackages is a tool to create the Packages file with, describing all packages that are present in the archive. dpkg-scanpackages needs to have some information about the archive structure; it needs to know the name of the tree to process ('contrib/binary-i386', for example), relative to the archive root, the location of the override file, and optionally a prefix to add to the Filename fields. The override file can override priority and section information specified in the 'control' file from a package itself.

Example 10-4. Example override entries

amiga-fdisk	required	admin
at	important	admin
cron	important	admin
debian-cd	optional	contrib/admin
xezmlm	optional	contrib/admin
xwatch	optional	contrib/admin
dnsutils	standard	non-US
ssh	standard	non-US

Example 10-5. A typical dpkg-scanpackages run

```

[klecker;~/public_html/vim]-3> dpkg-scanpackages . /dev/null | tee Packages | gzip -9 > Packages.
** Packages in archive but missing from override file: **
  vim

Wrote 1 entries to output Packages file.

```

Chapter 11. Maintainership

11.1. Being a responsible package maintainer

A responsible package maintainer keeps his packages up to date, fixes bugs, and keeps the package up-to-date, and compliant with the latest Debian standards.

Whenever somebody files a bugreport, the bugreport is filed, along with any further information that is submitted to it. Bugs can be controlled through the email-interface at bugs.debian.org, and viewed through the web-based interface at <http://bugs.debian.org/>. The maintainer also has to keep good contacts with the upstream author of the package, submit bugs, and send in patches.

11.2. Red tape

The Debian New Maintainer process is a series of required proceedings to become a Debian developer. The process involves an identity check, a philosophy check, and a tasks-and-skills check. After passing all these, a Developer can start working on his own packages.

When a developer for some reason does not keep his package(s) up to speed, another developer can do an NMU (Non Maintainer Upload), fixing the package, and entering it into the Debian archive.

The WNPP (Work-Needing and Prospective Packages list) contains just that. Maintainers can check this list for packages they want to work on – or to check if the package they want to work on is already being worked on, but has not been entered in the archive yet. The WNPP list is controlled by a couple of tags, specifying the state of a package:

RFP
Request for package

ITP
intent to package

RFA
Request for adoption

ITA
Intent To Adopt

ITO
Intent To Orphan

Debian for developers bibliography

[DebianPolicy] Ian Jackson, Christian Schwarz, and David A. Morris, 1996, *Debian policy manual*.

[FHS22] Daniel Quinlan, 1994–2000, *Filesystem Hierarchy Standard 2.2*.