Debian New Maintainers’ Guide

Josip Rodin and Osamu Aoki
This document may be used under the terms of the GNU General Public License version 2 or higher.

This document was made using these two documents as examples:

## COLLABORATORS

<table>
<thead>
<tr>
<th>ACTION</th>
<th>NAME</th>
<th>DATE</th>
<th>SIGNATURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITTEN BY</td>
<td>Josip Rodin and Osamu Aoki</td>
<td>December 24, 2020</td>
<td></td>
</tr>
</tbody>
</table>

## REVISION HISTORY

<table>
<thead>
<tr>
<th>NUMBER</th>
<th>DATE</th>
<th>DESCRIPTION</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Contents

1 Getting started The Right Way 1  
1.1 Social dynamics of Debian .................................................. 1  
1.2 Programs needed for development ........................................... 3  
1.3 Documentation needed for development .................................... 4  
1.4 Where to ask for help .......................................................... 4

2 First steps 6  
2.1 Debian package building workflow ............................................ 6  
2.2 Choose your program ......................................................... 7  
2.3 Get the program, and try it out ............................................... 9  
2.4 Simple build systems .......................................................... 10  
2.5 Popular portable build systems .............................................. 10  
2.6 Package name and version .................................................... 11  
2.7 Setting up dh_make ............................................................ 11  
2.8 Initial non-native Debian package .......................................... 12

3 Modifying the source 13  
3.1 Setting up quilt ............................................................... 13  
3.2 Fixing upstream bugs ........................................................ 13  
3.3 Installation of files to their destination .................................... 14  
3.4 Differing libraries .............................................................. 16

4 Required files under the debian directory 17  
4.1 control .......................................................................... 17  
4.2 copyright .................................................................... 21  
4.3 changelog ................................................................... 22  
4.4 rules ......................................................................... 23  
4.4.1 Targets of the rules file .................................................. 23  
4.4.2 Default rules file ........................................................ 24  
4.4.3 Customization of rules file ............................................. 26
## Other files under the debian directory

<table>
<thead>
<tr>
<th>Section</th>
<th>File</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>README.Debian</td>
<td>29</td>
</tr>
<tr>
<td>5.2</td>
<td>compat</td>
<td>30</td>
</tr>
<tr>
<td>5.3</td>
<td>conffiles</td>
<td>30</td>
</tr>
<tr>
<td>5.4</td>
<td>package.cron.*</td>
<td>30</td>
</tr>
<tr>
<td>5.5</td>
<td>dirs</td>
<td>31</td>
</tr>
<tr>
<td>5.6</td>
<td>package.doc-base</td>
<td>31</td>
</tr>
<tr>
<td>5.7</td>
<td>docs</td>
<td>31</td>
</tr>
<tr>
<td>5.8</td>
<td>emacsen-*</td>
<td>31</td>
</tr>
<tr>
<td>5.9</td>
<td>package.examples</td>
<td>32</td>
</tr>
<tr>
<td>5.10</td>
<td>package.init and package.default</td>
<td>32</td>
</tr>
<tr>
<td>5.11</td>
<td>install</td>
<td>32</td>
</tr>
<tr>
<td>5.12</td>
<td>package.info</td>
<td>32</td>
</tr>
<tr>
<td>5.13</td>
<td>package.links</td>
<td>33</td>
</tr>
<tr>
<td>5.14</td>
<td>{package,.source/}lintian-overrides</td>
<td>33</td>
</tr>
<tr>
<td>5.15</td>
<td>manpage.*</td>
<td>33</td>
</tr>
<tr>
<td>5.15.1</td>
<td>manpage.1.ex</td>
<td>33</td>
</tr>
<tr>
<td>5.15.2</td>
<td>manpage.sgml.ex</td>
<td>34</td>
</tr>
<tr>
<td>5.15.3</td>
<td>manpage.xml.ex</td>
<td>34</td>
</tr>
<tr>
<td>5.16</td>
<td>package.manpages</td>
<td>34</td>
</tr>
<tr>
<td>5.17</td>
<td>NEWS</td>
<td>34</td>
</tr>
<tr>
<td>5.18</td>
<td>{pre,post}{inst,rm}</td>
<td>35</td>
</tr>
<tr>
<td>5.19</td>
<td>package.symbols</td>
<td>35</td>
</tr>
<tr>
<td>5.20</td>
<td>TODO</td>
<td>35</td>
</tr>
<tr>
<td>5.21</td>
<td>watch</td>
<td>35</td>
</tr>
<tr>
<td>5.22</td>
<td>source/format</td>
<td>36</td>
</tr>
<tr>
<td>5.23</td>
<td>source/local-options</td>
<td>36</td>
</tr>
<tr>
<td>5.24</td>
<td>source/options</td>
<td>36</td>
</tr>
<tr>
<td>5.25</td>
<td>patches/*</td>
<td>37</td>
</tr>
</tbody>
</table>

## Building the package

<table>
<thead>
<tr>
<th>Section</th>
<th>Command or Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Complete (re)build</td>
<td>38</td>
</tr>
<tr>
<td>6.2</td>
<td>Autobuilder</td>
<td>39</td>
</tr>
<tr>
<td>6.3</td>
<td>debuild command</td>
<td>40</td>
</tr>
<tr>
<td>6.4</td>
<td>pbuilder package</td>
<td>40</td>
</tr>
<tr>
<td>6.5</td>
<td>git-buildpackage command and similar</td>
<td>42</td>
</tr>
<tr>
<td>6.6</td>
<td>Quick rebuild</td>
<td>42</td>
</tr>
<tr>
<td>6.7</td>
<td>Command hierarchy</td>
<td>43</td>
</tr>
</tbody>
</table>
7 Checking the package for errors

7.1 Suspicious changes ................................................. 44
7.2 Verifying a package's installation ................................. 44
7.3 Verifying a package's maintainer scripts ......................... 44
7.4 Using lintian ......................................................... 45
7.5 The debc command .................................................. 46
7.6 The debdiff command .............................................. 46
7.7 The interdiff command ............................................ 46
7.8 The mc command ..................................................... 46

8 Updating the package

8.1 New Debian revision ................................................ 47
8.2 Inspection of the new upstream release ......................... 48
8.3 New upstream release .............................................. 48
8.4 Updating the packaging style .................................... 49
8.5 UTF-8 conversion .................................................. 50
8.6 Reminders for updating packages ............................... 50

9 Uploading the package

9.1 Uploading to the Debian archive ................................. 51
9.2 Including orig.tar.gz for upload ................................ 52
9.3 Skipped uploads ..................................................... 52

A Advanced packaging

A.1 Shared libraries ..................................................... 53
A.2 Managing debian/package.symbols ............................ 54
A.3 Multiarch ............................................................ 55
A.4 Building a shared library package .............................. 56
A.5 Native Debian package ............................................ 57
Chapter 1

Getting started The Right Way

This document tries to describe the building of a Debian package to ordinary Debian users and prospective developers. It uses fairly non-technical language, and it’s well covered with working examples. There is an old Latin saying: *Longum iter est per praecepta, breve et efficax per exempla* (It’s a long way by the rules, but short and efficient with examples).

The rewrite of this tutorial document with updated contents and more practical examples is available as Guide for Debian Maintainers ([https://www.debian.org/doc/devel-manuals#debmake-doc](https://www.debian.org/doc/devel-manuals#debmake-doc)). Please use this new tutorial as the primary tutorial document.

This document is made available for the Debian Buster release since this offers many translations. This document will be dropped in the following releases since contents are getting outdated. ¹

One of the things that makes Debian such a top-notch distribution is its package system. While there is a vast quantity of software already in the Debian format, sometimes you need to install software that isn’t. You may be wondering how you can make your own packages; and perhaps you think it is a very difficult task. Well, if you are a real novice on Linux, it is hard, but if you were a rookie, you wouldn’t be reading this document now :-) You do need to know a little about Unix programming but you certainly don’t need to be a wizard. ²

One thing is certain, though: to properly create and maintain Debian packages takes many hours. Make no mistake, for our system to work the maintainers need to be both technically competent and diligent.

If you need some help with packaging, please read Section 1.4.

Newer versions of this document should always be available online at [http://www.debian.org/doc/maint-guide/](http://www.debian.org/doc/maint-guide/) and in the maint-guide package. The translations may be available in packages such as maint-guide-es. Please note that this documentation may be slightly outdated.

Since this is a tutorial, I choose to explain each detailed step for some important topics. Some of them may look irrelevant to you. Please be patient. I have also intentionally skipped some corner cases and provided only pointers to keep this document simple.

### 1.1 Social dynamics of Debian

Here are some observations of Debian’s social dynamics, presented in the hope that it will prepare you for interactions with Debian:

- We all are volunteers.
  - You cannot impose on others what to do.

---

¹The document assumes you are using a jessie or newer system. If you need to follow this text in an older system (including an older Ubuntu system etc.), you must install backported dpkg and debhelper packages, at least.

²You can learn about the basic handling of a Debian system from the Debian Reference ([http://www.debian.org/doc/manuals/debian-reference/](http://www.debian.org/doc/manuals/debian-reference/)). It contains some pointers to learn about Unix programming, too.
– You should be motivated to do things by yourself.

• Friendly cooperation is the driving force.
  – Your contribution should not overstrain others.
  – Your contribution is valuable only when others appreciate it.

• Debian is not your school where you get automatic attention of teachers.
  – You should be able to learn many things by yourself.
  – Attention from other volunteers is a very scarce resource.

• Debian is constantly improving.
  – You are expected to make high quality packages.
  – You should adapt yourself to change.

There are several types of people interacting around Debian with different roles:

• upstream author: the person who made the original program.

• upstream maintainer: the person who currently maintains the program.

• maintainer: the person making the Debian package of the program.

• sponsor: a person who helps maintainers to upload packages to the official Debian package archive (after checking their contents).

• mentor: a person who helps novice maintainers with packaging etc.

• Debian Developer (DD): a member of the Debian project with full upload rights to the official Debian package archive.

• Debian Maintainer (DM): a person with limited upload rights to the official Debian package archive.

Please note that you cannot become an official Debian Developer (DD) overnight, because it takes more than technical skill. Please do not be discouraged by this. If it is useful to others, you can still upload your package either as a maintainer through a sponsor or as a Debian Maintainer.

Please note that you do not need to create any new package to become an official Debian Developer. Contributing to the existing packages can provide a path to becoming an official Debian Developer too. There are many packages waiting for good maintainers (see Section 2.2).

Since we focus only on technical aspects of packaging in this document, please refer to the following to learn how Debian functions and how you can get involved:


• How can you help Debian? (http://www.debian.org/intro/help) (official)

• The Debian GNU/Linux FAQ, Chapter 13 - "Contributing to the Debian Project" (https://www.debian.org/doc/manuals/debian-faq/contributing.en.html) (semi-official)

• Debian Wiki, HelpDebian (http://wiki.debian.org/HelpDebian) (supplemental)

• Debian New Member site (https://nm.debian.org/) (official)

• Debian Mentors FAQ (http://wiki.debian.org/DebianMentorsFaq) (supplemental)
1.2 Programs needed for development

Before you start anything, you should make sure that you have properly installed some additional packages needed for development. Note that the list doesn’t contain any packages marked essential or required - we expect that you have those installed already.

The following packages come with the standard Debian installation, so you probably have them already (along with any additional packages they depend on). Still, you should check them with `aptitude show package` or with `dpkg -s package`.

The most important package to install on your development system is the build-essential package. Once you try to install that, it will pull in other packages required to have a basic build environment.

For some types of packages, that is all you will require; however, there is another set of packages that while not essential for all package builds are useful to have installed or may be required by your package:

- autoconf, automake, and autotools-dev - many newer programs use configure scripts and Makefile files preprocessed with the help of programs like these (see `info autoconf`, `info automake`). autotools-dev keeps up-to-date versions of certain auto files and has documentation about the best way to use those files.

- debhelper and dh-make - dh-make is necessary to create the skeleton of our example package, and it will use some of the debhelper tools for creating packages. They are not essential for this purpose, but are highly recommended for new maintainers. It makes the whole process much easier to start, and to control afterwards. (See `dh_make(8)`, `debhelper(1)`.)

The new debmake may be used as the alternative to the standard dh-make. It does more and comes with HTML documentation with extensive packaging examples in `debmake-doc`.

- devscripts - this package contains some useful scripts that can be helpful for maintainers, but they are also not necessary for building packages. Packages recommended and suggested by this package are worth looking into. (See `/usr/share/doc/devscripts/README.gz`.)

- fakeroot - this utility lets you emulate being root, which is necessary for some parts of the build process. (See `fakeroot(1)`.)

- file - this handy program can determine what type a file is. (See `file(1)`.)

- gfortran - the GNU Fortran 95 compiler, necessary if your program is written in Fortran. (See `gfortran(1)`.)

- git - this package provides a popular version control system designed to handle very large projects with speed and efficiency; it is used for many high profile open source projects, most notably the Linux kernel. (See `git(1)`). git Manual (`/usr/share/doc/git-doc/index.html`).

- gnupg - a tool that enables you to digitally sign packages. This is especially important if you want to distribute packages to other people, and you will certainly be doing that when your work gets included in the Debian distribution. (See `gpg(1)`.)

- gpc - the GNU Pascal compiler, necessary if your program is written in Pascal. Worthy of note here is `fp-compiler`, the Free Pascal Compiler, which is also good at this task. (See `gpc(1)`, `ppc386(1)`.)

- lintian - this is the Debian package checker, which lets you know of any common mistakes after you build the package and explains the errors found. (See `lintian(1)`, `Lintian User’s Manual (https://lintian.debian.org/manual/index.html).`)

- patch - this very useful utility will take a file containing a difference listing (produced by the `diff` program) and apply it to the original file, producing a patched version. (See `patch(1)`.)

- patchutils - this package contains some utilities to work with patches such as the `lsdiff`, `interdiff` and `filterdiff` commands.

- pbuilder - this package contains programs which are used for creating and maintaining a `chroot` environment. Building a Debian package in this `chroot` environment verifies the proper build dependency and avoids FTBFS (Fails To Build From Source) bugs. (see `pbuilder(8)` and `pdebuild(1)`)

- perl - Perl is one of the most used interpreted scripting languages on today’s Unix-like systems, often referred to as Unix’s Swiss Army Chainsaw. (See `perl(1)`.)

3There are also some more specialized but similar packages such as `dh-make-perl`, `dh-make-php`, etc.
• **python** - Python is another of the most used interpreted scripting languages on the Debian system, combining remarkable power with very clear syntax. (See `python(1)`.)

• **quilt** - this package helps you to manage large numbers of patches by keeping track of the changes each patch makes. Patches can be applied, un-applied, refreshed, and more. (See `quilt(1)`, and `~/usr/share/doc/quilt/quilt.pdf.gz`.)

• **xutils-dev** - some programs, usually those made for X11, also use these programs to generate Makefile files from sets of macro functions. (See `imake(1)`, `xmkmf(1)`.)

The short descriptions that are given above only serve to introduce you to what each package does. Before continuing please read the documentation of each relevant program including ones installed through the package dependency such as `make`, at least, for the standard usage. It may seem like heavy going now, but later on you'll be very glad you read it. If you have specific questions later, I would suggest re-reading the documents mentioned above.

### 1.3 Documentation needed for development

The following is the very important documentation which you should read along with this document:

• **debian-policy** - the [Debian Policy Manual](http://www.debian.org/doc/devel-manuals#policy) includes explanations of the structure and contents of the Debian archive, several OS design issues, the Filesystem Hierarchy Standard ([http://www.debian.org/doc/packaging-manuals/fhs/fhs-3.0.html](http://www.debian.org/doc/packaging-manuals/fhs/fhs-3.0.html)) (FHS, which says where each file and directory should be), etc. For you, the most important thing is that it describes requirements that each package must satisfy to be included in the distribution. (See the local copies of `~/usr/share/doc/debian-policy/policy.pdf.gz` and `~/usr/share/doc/debian-policy/fhs/fhs-3.0.pdf.gz`.)

• **developers-reference** - the [Debian Developer’s Reference](http://www.debian.org/doc/devel-manuals#devref) describes all matters not specifically about the technical details of packaging, like the structure of the archive, how to rename, orphan, or adopt packages, how to do NMUs, how to manage bugs, best packaging practices, when and where to upload, etc. (See the local copy of `~/usr/share/doc/developers-reference/developers-reference.pdf`.)

The following is the important documentation which you should read along with this document:

• **Autotools Tutorial** ([http://www.lrde.epita.fr/~adl/autotools.html](http://www.lrde.epita.fr/~adl/autotools.html)) provides a very good tutorial for the GNU Build System known as the GNU Autotools, whose most important components are Autoconf, Automake, Libtool, and gettext.

• **gnu-standards** - this package contains two pieces of documentation from the GNU project: GNU Coding Standards ([http://www.gnu.org/prep/standards/html_node/index.html](http://www.gnu.org/prep/standards/html_node/index.html)), and Information for Maintainers of GNU Software ([http://www.gnu.org/-prepmaintain/html_node/index.html](http://www.gnu.org/-prepmaintain/html_node/index.html)). Although Debian does not require these to be followed, these are still helpful as guidelines and common sense. (See the local copies of `~/usr/share/doc/gnu-standards/standards.pdf.gz` and `~/usr/share/doc/gnu-standards/maintain.pdf.gz`.)

If this document contradicts any of the documents mentioned above, they are correct. Please file a bug report on the `maint-guide` package using `reportbug`.

The following is an alternative tutorial document that you may read along with this document:


### 1.4 Where to ask for help

Before you decide to ask your question in some public place, please read this fine documentation:

• files in `~/usr/share/doc/package` for all pertinent packages
• contents of `man command` for all pertinent commands
• contents of `info command` for all pertinent commands
• contents of debian-mentors@lists.debian.org mailing list archive (http://lists.debian.org/debian-mentors/)
• contents of debian-devel@lists.debian.org mailing list archive (http://lists.debian.org/debian-devel/)

You can use web search engines more effectively by including search strings such as `site:lists.debian.org` to limit the domain.

Making a small test package is a good way to learn details of packaging. Inspecting existing well maintained packages is the best way to learn how other people make packages.

If you still have questions about packaging that you couldn’t find answers to in the available documentation and web resources, you can ask them interactively:

• debian-mentors@lists.debian.org mailing list (http://lists.debian.org/debian-mentors/). (This mailing list is for the novice.)
• debian-devel@lists.debian.org mailing list (http://lists.debian.org/debian-devel/). (This mailing list is for the expert.)
• IRC (http://www.debian.org/support#irc) such as #debian-mentors.
• Teams focusing on a specific set of packages. (Full list at https://wiki.debian.org/Teams )
• Language-specific mailing lists such as debian-devel-{french,italian,portuguese,spanish}@lists.debian.org or debian-devel@debian.or.jp. (Full listing at https://lists.debian.org/devel.html (https://lists.debian.org/devel.html) and https://lists.debian.org/users.html (https://lists.debian.org/users.html) )

The more experienced Debian developers will gladly help you, if you ask properly after making your required efforts.

When you receive a bug report (yes, actual bug reports!), you will know that it is time for you to dig into the Debian Bug Tracking System (http://www.debian.org/Bugs/) and read the documentation there, to be able to deal with the reports efficiently. I highly recommend reading the Debian Developer’s Reference, 5.8. "Handling bugs" (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#bug-handling).

Even if it all worked well, it’s time to start praying. Why? Because in just a few hours (or days) users from all around the world will start to use your package, and if you made some critical error you’ll get mailbombed by numerous angry Debian users… Just kidding. :-)

Relax and be ready for bug reports, because there is a lot more work to be done before your package will be fully in line with Debian policies and its best practice guidelines (once again, read the real documentation for details). Good luck!
Chapter 2

First steps

Let's start by creating a package of your own (or, even better, adopting an existing one).

2.1 Debian package building workflow

If you are making a Debian package with an upstream program, the typical workflow of Debian package building involves generating several specifically named files for each step as follows:

• Get a copy of the upstream software, usually in a compressed tar format.
  – *package*-version.tar.gz

• Add Debian-specific packaging modifications to the upstream program under the `debian` directory, and create a non-native source package (that is, the set of input files used for Debian package building) in `3.0 (quilt)` format.
  – *package*_version.orig.tar.gz

  – *package*_version-revision.debian.tar.gz

  – *package*_version-revision.dsc

• Build Debian binary packages, which are ordinary installable package files in `.deb` format (or `.udeb` format, used by the Debian Installer) from the Debian source package.
  – *package*_version-revision_arch.deb

Please note that the character separating `package` and `version` was changed from `-` (hyphen) in the tarball name to `_` (underscore) in the Debian package filenames.

In the file names above, replace the `package` part with the **package name**, the `version` part with the **upstream version**, the `revision` part with the **Debian revision**, and the `arch` part with the **package architecture**, as defined in the Debian Policy Manual. 

Each step of this outline is explained with detailed examples in later sections.

---

1For the older style of non-native Debian source packages in `1.0` format, `package_version-revision.diff.gz` is used instead.

2.2 Choose your program

You have probably chosen the package you want to create. The first thing you need to do is check if the package is in the distribution archive already by using the following:

• the `aptitude` command
• the Debian packages (http://www.debian.org/distrib/packages) web page
• the Debian Package Tracker (https://tracker.debian.org/) web page

If the package already exists, well, install it! :-) If it happens to be orphaned (that is, if its maintainer is set to Debian QA Group (http://qa.debian.org/)), you may be able to pick it up if it’s still available. You may also adopt a package whose maintainer has filed a Request for Adoption (RFA).³

There are several package ownership status resources:

• The `wnpp-alert` command from the `devscripts` package
• Work-Needing and Prospective Packages (http://www.debian.org/devel/wnpp/)
• Debian Bug report logs: Bugs in pseudo-package wnpp in unstable (http://bugs.debian.org/wnpp)
• Debian Packages that Need Lovin’ (http://wnpp.debian.net/)
• Browse wnpp bugs based on debtags (http://wnpp-by-tags.debian.net/)

As a side note, it’s important to point out that Debian already has packages for most kinds of programs, and the number of packages already in the Debian archive is much larger than that of contributors with upload rights. Thus, contributions to packages already in the archive are far more appreciated (and more likely to receive sponsorship) by other developers ⁴. You can contribute in various ways:

• taking over orphaned, yet actively used, packages
• joining packaging teams (http://wiki.debian.org/Teams)
• triaging bugs of very popular packages
• preparing QA or NMU uploads (http://www.debian.org/doc/developers-reference/pkgs.html#nmu-qa-upload)

If you are able to adopt the package, get the sources (with something like `apt-get source packagename`) and examine them. This document unfortunately doesn’t include comprehensive information about adopting packages. Thankfully you shouldn’t have a hard time figuring out how the package works since someone has already done the initial setup for you. Keep reading, though; a lot of the advice below will still be applicable to your case.

If the package is new, and you decide you’d like to see it in Debian, proceed as follows:

• First, you must know that the program works, and have tried it for some time to confirm its usefulness.
• You must check that no one else is already working on the package on the Work-Needing and Prospective Packages (http://www.debian.org/devel/wnpp/) site. If no one else is working on it, file an ITP (Intent To Package) bug report to the wnpp pseudo-package using `reportbug`. If someone’s already on it, contact them if you feel you need to. If not — find another interesting program that nobody is maintaining.
• The software must have a license.

⁴Having said that, there will of course always be new programs that are worth packaging.
– For the main section, Debian Policy requires it to be fully compliant with the Debian Free Software Guidelines (DFSG (http://www.debian.org/social_contract#guidelines) ) and not to require a package outside of main for compilation or execution. This is the desired case.
– For the contrib section, it must comply with the DFSG but it may require a package outside of main for compilation or execution.
– For the non-free section, it may be non-compliant with the DFSG but it must be distributable.
– If you are unsure about where it should go, post the license text on debian-legal@lists.debian.org (http://lists.debian.org/debian-legal/) and ask for advice.

• The program should not introduce security and maintenance concerns into the Debian system.
  – The program should be well documented and its code needs to be understandable (i.e., not obfuscated).
  – You should contact the program’s author(s) to check if they agree with packaging it and are amicable to Debian. It is important to be able to consult with the author(s) in case of any problems with the program, so don’t try to package unmaintained software.
  – The program certainly should not run setuid root, or even better, it shouldn’t need to be setuid or setgid to anything.
  – The program should not be a daemon, or go in an */sbin directory, or open a port as root.

Of course, the last one is just a safety measure, and is intended to save you from enraging users if you do something wrong in some setuid daemon… When you gain more experience in packaging, you’ll be able to package such software.

As a new maintainer, you are encouraged to get some experience in packaging with easier packages and discouraged from creating complicated packages.

• Simple packages
  – single binary package, arch = all (collection of data such as wallpaper graphics)
  – single binary package, arch = all (executable written in an interpreted language such as POSIX shell)

• Intermediate complexity packages
  – single binary package, arch = any (ELF binary executables compiled from languages such as C and C++)
  – multiple binary packages, arch = any + all (packages for ELF binary executables + documentation)
  – upstream source in a format other than tar.gz or tar.bz2
  – upstream source containing undistributable contents

• High complexity packages
  – interpreter module package used by other packages
  – generic ELF library package used by other packages
  – multiple binary packages including an ELF library package
  – source package with multiple upstream sources
  – kernel module packages
  – kernel patch packages
  – any package with non-trivial maintainer scripts

Packaging high complexity packages is not too hard, but it requires a bit more knowledge. You should seek specific guidance for every complex feature. For example, some languages have their own sub-policy documents:

• Perl policy (http://www.debian.org/doc/packaging-manuals/perl-policy/)
• Python policy (http://www.debian.org/doc/packaging-manuals/python-policy/)
• Java policy (http://www.debian.org/doc/packaging-manuals/java-policy/)
There is another old Latin saying: *fabricando fit faber* (practice makes perfect). It is highly recommended to practice and experiment with all the steps of Debian packaging with simple packages while reading this tutorial. A trivial upstream tarball, `hello-sh-1.0.tar.gz`, created as follows may offer a good starting point:

```bash
$ mkdir -p hello-sh/hello-sh-1.0; cd hello-sh/hello-sh-1.0
$ cat > hello <<EOF
#!/bin/sh
# (C) 2011 Foo Bar, GPL2+
echo "Hello!"
EOF
$ chmod 755 hello
$ cd..
$ tar -cvzf hello-sh-1.0.tar.gz hello-sh-1.0
```

### 2.3 Get the program, and try it out

So the first thing to do is to find and download the original source code. Presumably you already have the source file that you picked up at the author’s homepage. Sources for free Unix programs usually come in `tar+gzip` format with the extension `.tar.gz`, `tar+bzip2` format with the extension `.tar.bz2`, or `tar+xz` format with the extension `.tar.xz`. These usually contain a directory called `package-version` with all the sources inside.

If the latest version of the source is available through a Version Control System (VCS) such as Git, Subversion, or CVS, you need to get it with `git clone`, `svn co`, or `cvs co` and repack it into `tar+gzip` format yourself by using the `--exclude-vcs` option.

If your program’s source comes as some other sort of archive (for instance, the filename ends in `.Z` or `.zip`), you should also unpack it with the appropriate tools and repack it.

If your program’s source comes with some contents which do not comply with DFSG, you should also unpack it to remove such contents and repack it with a modified upstream version containing `dfsg`.

As an example, I’ll use a program called `gentoo`, a GTK+ file manager.  

Create a subdirectory under your home directory named `debian` or `deb` or anything you find appropriate (e.g. just `~/gentoo` would do fine in this case). Place the downloaded archive in it, and extract it (with `tar xzf gentoo-0.9.12.tar.gz`). Make sure there are no warning messages, even `irrelevant` ones, because other people's unpacking tools may or may not ignore these anomalies, so they may have problems unpacking them. Your shell command line may look something like this:

```bash
$ mkdir ~/gentoo; cd ~/gentoo
$ wget http://www.example.org/gentoo-0.9.12.tar.gz
$ tar xzvf gentoo-0.9.12.tar.gz
$ ls -F
  gentoo-0.9.12/
    gentoo-0.9.12.tar.gz
```

Now you have another subdirectory, called `gentoo-0.9.12`. Change to that directory and thoroughly read the provided documentation. Usually there are files named `README*`, `INSTALL*`, `*.lsm` or `*.html`. You must find instructions on how to compile and install the program (most probably they’ll assume you want to install to the `/usr/local/bin` directory; you won’t be doing that, but more on that later in Section 3.3).

You should start packaging with a completely clean (pristine) source directory, or simply with freshly unpacked sources.

---

5 Do not worry about the missing `Makefile`. You can install the `hello` command by simply using `debhelper` as in Section 5.11, or by modifying the upstream source to add a new `Makefile` with the `install` target as in Chapter 3.

6 You can identify the archive format using the `file` command when the file extension is not enough.

7 This program is already packaged. The current version (http://packages.qa.debian.org/g/gentoo.html) uses Autotools as its build structure and is substantially different from the following examples, which were based on version 0.9.12.
2.4 Simple build systems

Simple programs usually come with a Makefile and can be compiled just by invoking make. Some of them support make check, which runs included self-tests. Installation to the destination directories is usually done with make install.

Now try to compile and run your program, to make sure it works properly and doesn’t break something else while it’s installing or running.

Also, you can usually run make clean (or better make distclean) to clean up the build directory. Sometimes there’s even a make uninstall which can be used to remove all the installed files.

2.5 Popular portable build systems

A lot of free software programs are written in the C and C++ languages. Many of these use Autotools or CMake to make them portable across different platforms. These build tools need to be used to generate the Makefile and other required source files first. Then, such programs are built using the usual make; make install.

Autotools is the GNU build system comprising Autoconf, Automake, Libtool, and gettext. You can recognize such sources by the configure.ac, Makefile.am, and Makefile.in files.  

The first step of the Autotools workflow is usually that upstream runs autoreconf -i -f in the source directory and distributes the generated files along with the source.

```
configure.ac------+.> autoreconf -i+-> configure
Makefile.am ------+                  ++-> Makefile.in
src/Makefile.am ++                ++-> src/Makefile.in
   |                        ++-> config.h.in
   |          automake
   |          aclocal
   |          aclocal.m4
   |          autoheader
```

Editing configure.ac and Makefile.am files requires some knowledge of autoconf and automake. See info autoconf and info automake. The second step of the Autotools workflow is usually that the user obtains this distributed source and runs ./configure && make in the source directory to compile the program into an executable command binary.

```
Makefile.in ------+.> Makefile ------+.> make -> binary
src/Makefile.in ++-> ./configure ++-> src/Makefile ++
config.h.in ------+.> config.h ------+.> config.status ++
config.guess ++
```

You can change many things in the Makefile; for instance you can change the default location for file installation using the option ./configure --prefix=/usr.

Although it is not required, updating the configure and other files with autoreconf -i -f may improve the compatibility of the source.

CMake is an alternative build system. You can recognize such sources by the CMakeLists.txt file.

---

8Many modern programs come with a script named configure, which when executed creates a Makefile customized for your system.

9Autotools is too big to deal with in this small tutorial. This section is meant to provide keywords and references only. Please make sure to read the Autotools Tutorial (http://www.lrde.epita.fr/~adl/autotools.html) and the local copy of /usr/share/doc/autotools-dev/README.Debian.gz, if you need to use it.

10You can automate this by using dh-autoreconf package. See Section 4.4.3.
2.6 Package name and version

If the upstream source comes as `gentoo-0.9.12.tar.gz`, you can take `gentoo` as the (source) package name and `0.9.12` as the upstream version. These are used in the `debian/changelog` file described later in Section 4.3, too.

Although this simple approach works most of the time, you may need to adjust package name and upstream version by renaming the upstream source to follow Debian Policy and existing convention.

You must choose the package name to consist only of lower case letters (a-z), digits (0-9), plus (+) and minus (-) signs, and periods (.). It must be at least two characters long, must start with an alphanumeric character, and must not be the same as existing packages. It is a good idea to keep its length within 30 characters.

If upstream uses some generic term such as test-suite for its name, it is a good idea to rename it to identify its contents explicitly and avoid namespace pollution.

You should choose the upstream version to consist only of alphanumerics (0-9A-Za-z), plus signs (+), tildes (~), and periods (.). It must start with a digit (0-9). It is good idea to keep its length within 8 characters if possible.

If upstream does not use a normal versioning scheme such as 2.30.32 but uses some kind of date such as 11Apr29, a random codename string, or a VCS hash value as part of the version, make sure to remove them from the upstream version. Such information can be recorded in the `debian/changelog` file. If you need to invent a version string, use the YYYYMMDD format such as 20110429 as upstream version. This ensures that dpkg interprets later versions correctly as upgrades. If you need to ensure smooth transition to the normal version scheme such as 0.1 in the future, use the 0~YYMMDD format such as 0~110429 as the upstream version.

Version strings can be compared using dpkg(1) as follows:

```
$ dpkg --compare-versions ver1 op ver2
```

The version comparison rule can be summarized as:

- Strings are compared from the head to the tail.
- Letters are larger than digits.
- Numbers are compared as integers.
- Letters are compared in ASCII code order.
- There are special rules for period (.), plus (+), and tilde (~) characters, as follows:

```
0.0 < 0.5 < 0.10 < 0.99 < 1 < 1.0~rc1 < 1.0 < 1.0+b1 < 1.0+nmu1 < 1.1 < 2.0
```

One tricky case occurs when upstream releases `gentoo-0.9.12-ReleaseCandidate-99.tar.gz` as the pre-release of `gentoo-0.9.12.tar.gz`. You need to make sure that the upgrade works properly by renaming the upstream source to `gentoo-0.9.12-rc99.tar.gz`.

2.7 Setting up dh_make

Set up the shell environment variables `$DEBEMAIL` and `$DEBFULLNAME` so that various Debian maintenance tools recognize your email address and name to use for packages. 

---

11 The default package name field length of aptitude is 30. For more than 90% of packages, the package name is less than 24 characters.

12 If you follow the Debian Developer’s Reference 5.1. “New packages” (http://www.debian.org/doc/developers-reference/pkgs.html#newpackage), the ITP process will usually catch this kind of issue.

13 This stricter rule should help you avoid confusing file names.

14 The default version field length of aptitude is 10. The Debian revision with preceding hyphen usually consumes 2. For more than 80% of packages, the upstream version is less than 8 characters and the Debian revision is less than 2 characters. For more than 90% of packages, the upstream version is less than 10 characters and the Debian revision is less than 3 characters.

15 Version strings may be upstream version (version), Debian revision (revision), or version (version-revision). See Section 8.1 for how the Debian revision is incremented.

16 The following text assumes you are using Bash as your login shell. If you use some other login shell such as Z shell, use their corresponding configuration files instead of ~/.bashrc.
2.8 Initial non-native Debian package

Normal Debian packages are non-native Debian packages made from upstream programs. If you wish to create a non-native Debian package of an upstream source `gentoo-0.9.12.tar.gz`, you can create an initial non-native Debian package for it by issuing the `dh_make` command as follows:

```
$ cd ~/gentoo
$ wget http://example.org/gentoo-0.9.12.tar.gz
$ tar -xvzf gentoo-0.9.12.tar.gz
$ cd gentoo-0.9.12
$ dh_make -f ../gentoo-0.9.12.tar.gz
```

Of course, replace the filename with the name of your original source archive. See `dh_make(8)` for details.

You should see some output asking you what sort of package you want to create. Gentoo is a single binary package—it creates only one binary package, i.e., one `.deb` file—so we will select the first option (with the `s` key), check the information on the screen, and confirm by pressing `ENTER`.

This execution of `dh_make` creates a copy of the upstream tarball as `gentoo_0.9.12.orig.tar.gz` in the parent directory to accommodate the creation of the non-native Debian source package with the name `debian.tar.gz` later:

```
$ cd ~/gentoo; ls -F
gentoo-0.9.12/
gentoo-0.9.12.tar.gz
gentoo-0.9.12.orig.tar.gz
```

Please note two key features of this filename `gentoo_0.9.12.orig.tar.gz`:

- Package name and version are separated by the character `_` (underscore).
- The string `.orig` is inserted before the `.tar.gz`.

You should also notice that many template files are created in the source under the `debian` directory. These will be explained in Chapter 4 and Chapter 5. You should also understand that packaging cannot be a fully automated process. You will need to modify the upstream source for Debian (see Chapter 3). After this, you need to use the proper methods for building Debian packages (Chapter 6), testing them (Chapter 7), and uploading them (Chapter 9). All the steps will be explained.

If you accidentally erased some template files while working on them, you can recover them by running `dh_make` with the `--addmissing` option again in a Debian package source tree.

Updating an existing package may get complicated since it may be using older techniques. While learning the basics, please stick to creating a fresh package; further explanations are given in Chapter 8.

Please note that the source file does not need to contain any build system discussed in Section 2.4 and Section 2.5. It could be just a collection of graphical data, etc. Installation of files may be carried out using only `debhelper` configuration files such as `debian/install` (see Section 5.11).

---

17 If the upstream source provides the `debian` directory and its contents, run the `dh_make` command with the extra option `--addmissing`. The new source 3.0 (quilt) format is robust enough not to break even for these packages. You may need to update the contents provided by the upstream version for your Debian package.

18 There are several choices here: `s` for Single binary package, `i` for arch-Independent package, `m` for Multiple binary packages, `l` for Library package, `k` for Kernel module package, `n` for kernel patch package, and `b` for `cdbs` package. This document focuses on the use of the `dh` command (from the package `debhelper`) to create a single binary package, but also touches on how to use it for arch-independent or multiple binary packages. The package `cdbs` offers an alternative packaging script infrastructure to the `dh` command and is outside the scope of this document.
Chapter 3

Modifying the source

Please note that there isn’t space here to go into all the details of fixing upstream sources, but here are some basic steps and problems people often run across.

3.1 Setting up quilt

The program `quilt` offers a basic method for recording modifications to the upstream source for Debian packaging. It’s useful to have a slightly customized default, so let’s create an alias `dquilt` for Debian packaging by adding the following lines to `~/.bashrc`. The second line provides the same shell completion feature of the `quilt` command to the `dquilt` command:

```
alias dquilt="quilt --quiltrc=${HOME}/.quiltrc-dpkg"
complete -F _quilt_completion -o filenames dquilt
```

Then let’s create `~/.quiltrc-dpkg` as follows:

```
d=.; while [ ! -d $d/debian -a $(readlink -e $d) != / ]; do d=$d/..; done
if [ -d $d/debian ] && [ -z $QUILT_PATCHES ]; then
  # if in Debian packaging tree with unset $QUILT_PATCHES
  QUILT_PATCHES="debian/patches"
  QUILT_PATCH_OPTS="--reject-format=unified"
  QUILT_DIFF_ARGS="-p ab --no-timestamps --no-index --color=auto"
  QUILT_REFRESH_ARGS="-p ab --no-timestamps --no-index"
  QUILT_COLORS="diff_hdr=1;32:diff_add=1;34:diff_rem=1;31:diff_hunk=1;33:diff_ctx=35:←diff_cctx=33"
  if ! [ -d $d/debian/patches ]; then mkdir $d/debian/patches; fi
fi
```

See `quilt(1)` and `/usr/share/doc/quilt/quilt.pdf.gz` on how to use `quilt`.

3.2 Fixing upstream bugs

Let’s assume you find an error in the upstream Makefile as follows, where `install: gentoo` should have been `install: gentoo-target`.

```
install: gentoo
    install ./gentoo $(BIN)
    install icons/* $(ICONS)
    install gentoorc-example $(HOME)/gentoorc
```
Let’s fix this and record it with the `dquilt` command as `fix-gentoo-target.patch`:

```bash
$ mkdir debian/patches
$ dquilt new fix-gentoo-target.patch
$ dquilt add Makefile
```

You change the `Makefile` file as follows:

```makefile
install: gentoo-target
    install ./gentoo $(BIN)
    install icons/* $(ICONS)
    install gentoorc-example $(HOME)/.gentoorc
```

Ask `dquilt` to generate the patch to create `debian/patches/fix-gentoo-target.patch` and add its description following DEP-3: Patch Tagging Guidelines (http://dep.debian.net/deps/dep3):

```bash
$ dquilt refresh
$ dquilt header -e
  ... describe patch
```

### 3.3 Installation of files to their destination

Most third-party software installs itself in the `/usr/local` directory hierarchy. On Debian this is reserved for private use by the system administrator, so packages must not use directories such as `/usr/local/bin` but should instead use system directories such as `/usr/bin`, obeying the Filesystem Hierarchy Standard (http://www.debian.org/doc/packaging-manuals/fhs/fhs-3.0.html) (FHS).

Normally, `make(1)` is used to automate building the program, and executing `make install` installs programs directly to the desired destination (following the `install` target in the `Makefile`). In order for Debian to provide pre-built installable packages, it modifies the build system to install programs into a file tree image created under a temporary directory instead of the actual destination.

These two differences between normal program installation on one hand and the Debian packaging system on the other can be transparently addressed by the `debhelper` package through the `dh_auto_configure` and `dh_auto_install` commands if the following conditions are met:

- The `Makefile` must follow GNU conventions and support the `$(DESTDIR)` variable.  
- The source must follow the Filesystem Hierarchy Standard (FHS).

Programs that use GNU `autoconf` follow the GNU conventions automatically, so they can be trivial to package. On the basis of this and other heuristics, it is estimated that the `debhelper` package will work for about 90% of packages without making any intrusive changes to their build system. So packaging is not as complicated as it may seem.

If you need to make changes in the `Makefile`, you should be careful to support the `$(DESTDIR)` variable. Although it is unset by default, the `$(DESTDIR)` variable is prepended to each file path used for the program installation. The packaging script will set `$(DESTDIR)` to the temporary directory.

For a source package generating a single binary package, the temporary directory used by the `dh_auto_install` command will be set to `debian/package`.  

---

1The `debian/patches` directory should exist now if you ran `dh_make` as described before. This example operation creates it just in case you are updating an existing package.


3For a source package generating multiple binary packages, the `dh_auto_install` command uses `debian/tmp` as the temporary directory while the `dh_install` command with the help of `debian/package-1.install` and `debian/package-2.install` files will split the contents of `debian/tmp` into `debian/package-1` and `debian/package-2` temporary directories, to create `package-1_*.deb` and `package-2_*.deb` binary packages.
they install your package; the only difference is that `dpkg` will be installing the files to paths relative to the root directory rather than your working directory.

Bear in mind that even though your program installs in `debian/package`, it still needs to behave correctly when installed from the `.deb` package under the root directory. So you must not allow the build system to hardcode strings like `/home/me/deb/package-version/usr/share/package` into files in the package.

Here's the relevant part of gentoo's `Makefile`:

```makefile
# Where to put executable commands on 'make install'?
BIN = /usr/local/bin
# Where to put icons on 'make install'?
ICONS = /usr/local/share/gentoo
```

We see that the files are set to install under `/usr/local`. As explained above, that directory hierarchy is reserved for local use on Debian, so change those paths as follows:

```makefile
# Where to put executable commands on 'make install'?
BIN = $(DESTDIR)/usr/bin
# Where to put icons on 'make install'?
ICONS = $(DESTDIR)/usr/share/gentoo
```

The exact locations that should be used for binaries, icons, documentation, etc. are specified in the Filesystem Hierarchy Standard (FHS). You should browse through it and read the sections relevant to your package.

So, we should install executable commands in `/usr/bin` instead of `/usr/local/bin`, the manual page in `/usr/share/man/man1` instead of `/usr/local/man/man1`, and so on. Notice how there's no manual page mentioned in gentoo's `Makefile`, but since Debian Policy requires that every program has one, we'll make one later and install it in `/usr/share/man/man1`.

Some programs don't use `Makefile` variables to define paths such as these. This means you might have to edit some real C sources in order to fix them to use the right locations. But where to search, and exactly what for? You can find this out by issuing:

```
$ grep -nr --include='*.c|h' -e 'usr/local/lib' .
```

`grep` will run recursively through the source tree and tell you the filename and the line number for all matches.

Edit those files and in those lines replace `usr/local/lib` with `usr/lib`. This can be done automatically as follows:

```
$ sed -i -e '/^#usr/local/lib/usr/lib/#' $(find . -type f -name '*.[c|h]')
```

If you want to confirm each substitution instead, this can be done interactively as follows:

```
$ vim '+'argdo %#usr/local/lib/usr/lib#gce|update' +q 
   $(find . -type f -name '*.[c|h]')
```

Next you should find the `install` target (searching for the line that starts with `install:` will usually work) and rename all references to directories other than ones defined at the top of the `Makefile`.

Originally, gentoo's `install` target said:

```makefile
install: gentoo-target
   install ./gentoo $(BIN)
   install icons/* $(ICONS)
   install gentoorc-example $(HOME)/.gentoorc
```

Let's fix this upstream bug and record it with the `dquilt` command as `debian/patches/install.patch`.

```
$ dquilt new install.patch
$ dquilt add Makefile
```

---

4This is just an example to show what a `Makefile` should look like. If the `Makefile` is created by the `./configure` command, the correct way to fix this kind of `Makefile` is to execute `./configure` from the `dh_auto_configure` command with default options including `--prefix=/usr`.

---
In your editor, change this for the Debian package as follows:

```bash
install: gentoo-target
install -d $(BIN) $(ICONS) $(DESTDIR)/etc
install ./gentoo $(BIN)
install -m644 icons/* $(ICONS)
install -m644 gentoorc-example $(DESTDIR)/etc/gentoorc
```

You’ll have noticed that there’s now an `install -d` command before the other commands in the rule. The original `Makefile` didn’t have it because usually `/usr/local/bin` and other directories already exist on the system where you are running `make install`. However, since we will be installing into a newly created private directory tree, we will have to create each and every one of those directories.

We can also add in other things at the end of the rule, like the installation of additional documentation that the upstream authors sometimes omit:

```bash
install -d $(DESTDIR)/usr/share/doc/gentoo/html
cp -a docs/* $(DESTDIR)/usr/share/doc/gentoo/html
```

Check carefully, and if everything is okay, ask `dquilt` to generate the patch to create `debian/patches/install.patch` and add its description:

```bash
$ dquilt refresh
$ dquilt header -e
... describe patch
```

Now you have a series of patches.

1. Upstream bug fix: `debian/patches/fix-gentoo-target.patch`
2. Debian specific packaging modification: `debian/patches/install.patch`

Whenever you make changes that are not specific to the Debian package such as `debian/patches/fix-gentoo-target.patch`, be sure to send them to the upstream maintainer so they can be included in the next version of the program and be useful to everyone else. Also remember to avoid making your fixes specific to Debian or Linux—or even Unix! Make them portable. This will make your fixes much easier to apply.

Note that you don’t have to send the `debian/*` files upstream.

### 3.4 Differing libraries

There is one other common problem: libraries are often different from platform to platform. For example, a `Makefile` can contain a reference to a library that doesn’t exist on the Debian system. In that case, we need to change it to a library that does exist in Debian, and serves the same purpose.

Let’s assume a line in your program’s `Makefile` (or `Makefile.in`) as the following.

```
LIBS = -lfoo -lbar
```

If your program doesn’t compile since the `foo` library doesn’t exist and its equivalent is provided by the `foo2` library on the Debian system, you can fix this build problem as `debian/patches/foo2.patch` by changing `foo` into `foo2`:

```bash
$ dquilt new foo2.patch
$ dquilt add Makefile
$ sed -i -e 's/-lfoo/-foo2/g' Makefile
$ dquilt refresh
$ dquilt header -e
... describe patch
```

If there are API changes from the `foo` library to the `foo2` library, required changes to the source code need to be made to match the new API.
Chapter 4

Required files under the debian directory

There is a new subdirectory under the program’s source directory, called debian. There are a number of files in this directory that we should edit in order to customize the behavior of the package. The most important of them are control, changelog, copyright, and rules, which are required for all packages. ¹

4.1 control

This file contains various values which dpkg, dselect, apt-get, apt-cache, aptitude, and other package management tools will use to manage the package. It is defined by the Debian Policy Manual, 5 “Control files and their fields” (http://www.debian.org/doc/debian-policy/ch-controlfields.html).

Here is the control file dh_make created for us:

```
1 Source: gentoo
2 Section: unknown
3 Priority: optional
4 Maintainer: Josip Rodin <joy-mg@debian.org>
5 Build-Depends: debhelper (>=10)
6 Standards-Version: 4.0.0
7 Homepage: <insert the upstream URL, if relevant>
8
9 Package: gentoo
10 Architecture: any
11 Depends: ${shlibs:Depends}, ${misc:Depends}
12 Description: <insert up to 60 chars description>
13 <insert long description, indented with spaces>
```

(I’ve added the line numbers.)

Lines 1–7 are the control information for the source package. Lines 9–13 are the control information for the binary package.

Line 1 is the name of the source package.

Line 2 is the section of the distribution the source package goes into.

As you may have noticed, the Debian archive is divided into multiple areas: main (the free software), non-free (the not really free software) and contrib (free software that depends on non-free software). Each of these is divided into sections that classify packages into rough categories. So we have admin for administrator-only programs, devel for programmer tools, doc for documentation, libs for libraries, mail for email readers and daemons, net for network apps and daemons, x11 for X11 programs that don’t fit anywhere else, and many more. ²

¹In this chapter, files in the debian directory are referred to without the leading debian/ for simplicity whenever the meaning is obvious.

²See Debian Policy Manual, 2.4 “Sections” (http://www.debian.org/doc/debian-policy/ch-archive.html#s-subsections) and List of sections in sid (http://packages.debian.org/unstable/).
Let’s change it then to x11. (A main/prefix is implied so we can omit it.)

Line 3 describes how important it is that the user installs this package.

- The optional priority will usually work for new packages that do not conflict with others claiming required, important, or standard priority.

Section and priority are used by front-ends like aptitude when they sort packages and select defaults. Once you upload the package to Debian, the value of these two fields can be overridden by the archive maintainers, in which case you will be notified by email.

As this is a normal priority package and doesn’t conflict with anything else, we will change the priority to optional.

Line 4 is the name and email address of the maintainer. Make sure that this field includes a valid To header for email, because after you upload it, the bug tracking system will use it to deliver bug emails to you. Avoid using commas, ampersands, or parentheses.

Line 5 includes the list of packages required to build your package as the Build-Depends field. You can also have the Build-Depends-Indep field as an additional line here.

- For all packages packaged with the dh command in the debian/rules file, you must have debhelper (>=9) in the Build-Depends field to satisfy the Debian Policy requirement for the clean target.

- Source packages which have binary packages with Architecture: any are rebuilt by the autobuilder. Since this autobuilder procedure installs only the packages listed in the Build-Depends field before running debian/rules build (see Section 6.2), the Build-Depends field needs to list practically all the required packages, and Build-Depends-Indep is rarely used.

- For source packages with binary packages all of which are Architecture: all, the Build-Depends-Indep field may list all the required packages unless they are already listed in the Build-Depends field to satisfy the Debian Policy requirement for the clean target.

If you are not sure which one should be used, use the Build-Depends field to be on the safe side.

To find out what packages your package needs to be built run the command:

$ dpkg-depcheck -d ./configure

To manually find exact build dependencies for /usr/bin/foo, execute

$ objdump -p /usr/bin/foo | grep NEEDED

and for each library listed (e.g., libfoo.so.6), execute

$ dpkg -S libfoo.so.6

Then just take the -dev version of every package as a Build-Depends entry. If you use ldd for this purpose, it will report indirect lib dependencies as well, resulting in the problem of excessive build dependencies.

gentoo also happens to require xlibs-dev, libgtk1.2-dev and libglib1.2-dev to build, so we’ll add them here next to debhelper.

---


5This somewhat strange situation is a feature well documented in the Debian Policy Manual, Footnotes 55 (http://www.debian.org/doc/debian-policy/-footnotes.html#f55). This is not due to the use of the dh command in the debian/rules file but due to how the dpkg-buildpackage works. The same situation applies to the auto build system for Ubuntu (https://bugs.launchpad.net/launchpad-buildd/+bug/238141).
Line 6 is the version of the Debian Policy Manual (http://www.debian.org/doc/devel-manuals#policy) standards this package follows, the one you read while making your package.

On line 7 you can put the URL of the software’s upstream homepage.

Line 9 is the name of the binary package. This is usually the same as the name of the source package, but it doesn’t necessarily have to be that way.

Line 10 describes the architectures the binary package can be compiled for. This value is usually one of the following depending on the type of the binary package:  

- **Architecture: any**
  - The generated binary package is an architecture dependent one usually in a compiled language.

- **Architecture: all**
  - The generated binary package is an architecture independent one usually consisting of text, images, or scripts in an interpreted language.

We leave line 10 as is since this is written in C. dpkg-gencontrol(1) will fill in the appropriate architecture value for any machine this source package gets compiled on.

If your package is architecture independent (for example, a shell or Perl script, or a document), change this to **all**, and read later in Section 4.4 about using the **binary-indep** rule instead of **binary-arch** for building the package.

Line 11 shows one of the most powerful features of the Debian packaging system. Packages can relate to each other in various ways. Apart from **Depends**, other relationship fields are **Recommends**, **Suggests**, **Pre-Depends**, **Breaks**, **Conflicts**, **Provides**, and **Replaces**.

The package management tools usually behave the same way when dealing with these relations; if not, it will be explained. (See dpkg(8), dselect(8), apt(8), aptitude(1), etc.)

Here is a simplified description of package relationships:

- **Depends**
  The package will not be installed unless the packages it depends on are installed. Use this if your program absolutely will not run (or will cause severe breakage) unless a particular package is present.

- **Recommends**
  Use this for packages that are not strictly necessary but are typically used with your program. When a user installs your program, all front-ends will probably prompt them to install the recommended packages. aptitude and apt-get install recommended packages along with your package by default (but the user can disable this behavior). dpkg will ignore this field.

- **Suggests**
  Use this for packages which will work nicely with your program but are not at all necessary. When a user installs your program, they will probably not be prompted to install suggested packages. aptitude can be configured to install suggested packages along with your package but this is not its default. dpkg and apt-get will ignore this field.

- **Pre-Depends**
  This is stronger than **Depends**. The package will not be installed unless the packages it pre-depends on are installed and correctly configured. Use this very sparingly and only after discussing it on the debian-devel@lists.debian.org (http://lists.debian.org/debian-devel) mailing list. Read: don’t use it at all. :-)

- **Conflicts**
  The package will not be installed until all the packages it conflicts with have been removed. Use this if your program absolutely will not run or will cause severe problems if a particular package is present.

---


• **Breaks**
  When installed the package will break all the listed packages. Normally a **Breaks** entry specifies that it applies to versions earlier than a certain value. The resolution is generally to use higher-level package management tools to upgrade the listed packages.

• **Provides**
  For some types of packages where there are multiple alternatives, virtual names have been defined. You can get the full list in the `virtual-package-names-list.txt.gz` (http://www.debian.org/doc/packaging-manuals/virtual-package-names-list.txt) file. Use this if your program provides a function of an existing virtual package.

• **Replaces**
  Use this when your program replaces files from another package, or completely replaces another package (used in conjunction with **Conflicts**). Files from the named packages will be overwritten with the files from your package.

All these fields have uniform syntax. They are a list of package names separated by commas. These package names may also be lists of alternative package names, separated by vertical bar symbols `|` (pipe symbols).

The fields may restrict their applicability to particular versions of each named package. The restriction of each individual package is listed in parentheses after its name, and should contain a relation from the list below followed by a version number value. The relations allowed are: `<<`, `<=`, `=`, `>=`, and `>>` for strictly lower, lower or equal, exactly equal, greater or equal, and strictly greater, respectively. For example,

```plaintext
Depends: foo (>= 1.2), libbar1 (= 1.3.4)
Conflicts: baz
Recommends: libbaz4 (>> 4.0.7)
Suggests: quux
Replaces: quux (< 5), quux-foo (<= 7.6)
```

The last feature you need to know about is `${shlibs:Depends}`, `${perl:Depends}`, `${misc:Depends}`, etc. `dh_shlibdeps(1)` calculates shared library dependencies for binary packages. It generates a list of ELF executables and shared libraries it has found for each binary package. This list is used for substituting `${shlibs:Depends}`.

`dh_perl(1)` calculates Perl dependencies. It generates a list of dependencies on `perl` or `perlapi` for each binary package. This list is used for substituting `${perl:Depends}`.

Some debhelper commands may cause the generated package to depend on some additional packages. All such commands generate a list of required packages for each binary package. This list is used for substituting `${misc:Depends}`.

`dh_gencontrol(1)` generates `DEBIAN/control` for each binary package while substituting `${shlibs:Depends}`, `${perl:Depends}`, `${misc:Depends}`, etc.

Having said all that, we can leave the **Depends** field exactly as it is now, and insert another line after it saying **Suggests:** file, because gentoo can use some features provided by the `file` package.

Line 9 is the Homepage URL. Let’s assume this to be at `http://www.obsession.se/gentoo/`.

Line 12 is the short description. Terminals are conventionally 80 columns wide so this shouldn’t be longer than about 60 characters. I’ll change it to fully GUI-configurable, two-pane X file manager.

Line 13 is where the long description goes. This should be a paragraph which gives more details about the package. Column 1 of each line should be empty. There must be no blank lines, but you can put a single . (dot) in a column to simulate that. Also, there must be no more than one blank line after the long description. 8

We can insert Vcs-* fields to document the Version Control System (VCS) location between lines 6 and 7. 9 Let’s assume that the gentoo package has its VCS located in the Debian Alioth Git Service at `git://git.debian.org/git/collab-maint/gentoo.git`.

Finally, here is the updated control file:
Copyright

This file contains information about the copyright and license of the upstream sources. Debian Policy Manual, 12.5 "Copyright information" (http://www.debian.org/doc/debian-policy/ch-docs.html#s-copyrightfile) dictates its content and DEP-5: Machine-parseable debian/copyright (http://dep.debian.net/deps/dep5/) provides guidelines for its format.

dh_make can give you a template copyright file. Let's use the --copyright gpl2 option here to get a template file for the gentoo package released under GPL-2.

You must fill in missing information to complete this file, such as the place you got the package from, the actual copyright notice, and the license. For certain common free software licenses (GNU GPL-1, GNU GPL-2, GNU GPL-3, LGPL-2, LGPL-2.1, LGPL-3, GNU FDL-1.2, GNU FDL-1.3, Apache-2.0, 3-Clause BSD, CC0-1.0, MPL-1.1, MPL-2.0 or the Artistic license), you can just refer to the appropriate file in the /usr/share/common-licenses/ directory that exists on every Debian system. Otherwise, you must include the complete license.

In short, here's what gentoo's copyright file should look like:

```
1 Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
2 Upstream-Name: gentoo
3 Upstream-Contact: Emil Brink <emil@obsession.se>
4 Source: http://sourceforge.net/projects/gentoo/files/
5
6 Files: *
7 Copyright: 1998-2010 Emil Brink <emil@obsession.se>
8 License: GPL-2+
9
10 Files: icons/*
11 Copyright: 1998 Johan Hanson <johan@tiq.com>
```
4.3 changelog

This is a required file, which has a special format described in Debian Policy Manual, 4.4 "debian/changelog" (http://www.debian.org/-doc/debian-policy/ch-source.html#s-dpkgchangelog). This format is used by dpkg and other programs to obtain the version number, revision, distribution, and urgency of your package.

For you, it is also important, since it is good to have documented all changes you have done. It will help people downloading your package to see whether there are issues with the package that they should know about. It will be saved as /usr/share/doc/gentoo/changelog.Debian.gz in the binary package.

dh_make created a default one, and this is what it looks like:

(I’ve added the line numbers.)

Please follow the HOWTO provided by the ftpmasters and sent to debian-devel-announce: http://lists.debian.org/debian-devel-announce/2006/03/msg00023.html.

Line 1 is the package name, version, distribution, and urgency. The name must match the source package name; distribution should be unstable, and urgency should be set to medium unless there is any particular reason for other values.

Lines 3-5 are a log entry, where you document changes made in this package revision (not the upstream changes — there is a special file for that purpose, created by the upstream authors, which you will later install as /usr/share/doc/gentoo/changelog.gz). Let’s assume your ITP (Intent To Package) bug report number was 12345. New lines must be inserted just below the uppermost line that begins with * (asterisk). You can do it with dch(1). You can edit this manually with a text editor as long as you follow the formatting convention used by the dch(1).
In order to prevent a package being accidentally uploaded before completing the package, it is a good idea to change the distribution value to an invalid distribution value of UNRELEASED.

You will end up with something like this:

```
1  gentoo (0.9.12-1) UNRELEASED; urgency=low
2  * Initial Release. Closes: #12345
3  * This is my first Debian package.
4  * Adjusted the Makefile to fix $(DESTDIR) problems.
5
6  -- Josip Rodin <joy-mg@debian.org> Mon, 22 Mar 2010 00:37:31 +0100
7
8 (I’ve added the line numbers.)
```

Once you are satisfied with all the changes and documented them in changelog, you should change the distribution value from UNRELEASED to the target distribution value unstable (or even experimental).\(^\text{10}\)

You can read more about updating the changelog file later in Chapter 8.

### 4.4 rules

Now we need to take a look at the exact rules that dpkg-buildpackage(1) will use to actually create the package. This file is in fact another Makefile, but different from the one(s) in the upstream source. Unlike other files in debian, this one is marked as executable.

#### 4.4.1 Targets of the rules file

Every rules file, like any other Makefile, consists of several rules, each of which defines a target and how it is carried out.\(^\text{11}\) A new rule begins with its target declaration in the first column. The following lines beginning with the TAB code (ASCII 9) specify the recipe for carrying out that target. Empty lines and lines beginning with # (hash) are treated as comments and ignored.\(^\text{12}\)

A rule that you want to execute is invoked by its target name as a command line argument. For example, `debian/rules build` and `fakeroot make -f debian/rules binary` execute rules for build and binary targets, respectively.

Here is a simplified explanation of the targets:

- **clean** target: to clean all compiled, generated, and useless files in the build-tree. (Required)
- **build** target: to build the source into compiled programs and formatted documents in the build-tree. (Required)
- **build-arch** target: to build the source into arch-dependent compiled programs in the build-tree. (Required)
- **build-indep** target: to build the source into arch-independent formatted documents in the build-tree. (Required)
- **install** target: to install files into a file tree for each binary package under the debian directory. If defined, binary* targets effectively depend on this target. (Optional)
- **binary** target: to create all binary packages (effectively a combination of binary-arch and binary-indep targets). (Required)\(^\text{13}\)

\(^\text{10}\)If you use the dch -r command to make this last change, please make sure to save the changelog file explicitly by the editor.


\(^\text{13}\)This target is used by dpkg-buildpackage as in Section 6.1.
• **binary-arch** target: to create arch-dependent (Architecture: any) binary packages in the parent directory. (Required)\textsuperscript{14}

• **binary-indep** target: to create arch-independent (Architecture: all) binary packages in the parent directory. (Required)\textsuperscript{15}

• **get-orig-source** target: to obtain the most recent version of the original source package from an upstream archive. (Optional)

You are probably overwhelmed by now, but things are much simpler upon examination of the rules file that \texttt{dh\_make} gives us as a default.

### 4.4.2 Default rules file

Newer \texttt{dh\_make} generates a very simple but powerful default rules file using the \texttt{dh} command:

```
1 #!/usr/bin/make -f
2 # See debhelper(7) (uncomment to enable)
3 # output every command that modifies files on the build system.
4 # DH\_VERBOSE = 1
5
6 # see FEATURE AREAS in dpkg\_buildflags(1)
7 #export DEB\_BUILD\_MAINT\_OPTIONS = hardening=+all
8
9 # see ENVIRONMENT in dpkg\_buildflags(1)
10 # package maintainers to append CFLAGS
11 #export DEB\_CFLAGS\_MAINT\_APPEND = -Wall -pedantic
12 # package maintainers to append LDFLAGS
13 #export DEB\_LDFLAGS\_MAINT\_APPEND = -Wl,--as-needed
14
15
16 %:
17   dh $@
```

(I’ve added the line numbers and trimmed some comments. In the actual rules file, the leading spaces are a TAB code.)

You are probably familiar with lines like line 1 from shell and Perl scripts. It tells the operating system that this file is to be processed with /usr/bin/make.

Line 4 can be uncommented to set the DH\_VERBOSE variable to 1, so that the \texttt{dh} command outputs which \texttt{dh\_} commands it is executing. You can also add a line \texttt{export DH\_OPTIONS=-v} here, so that each \texttt{dh\_} command outputs which commands it is executing. This helps you to understand exactly what is going on behind this simple rules file and to debug its problems.

This new \texttt{dh} is designed to form a core part of the debhelper tools, and not to hide anything from you.

Lines 16 and 17 are where all the work is done with an implicit rule using the pattern rule. The percent sign means "any targets", which then call a single program, \texttt{dh}, with the target name.\textsuperscript{16} The \texttt{dh} command is a wrapper script that runs appropriate sequences of \texttt{dh\_} programs depending on its argument.\textsuperscript{17}

• **debian/rules clean** runs \texttt{dh clean}, which in turn runs the following:

```
dh_testdir
dh_auto_clean
dh_clean
```

\textsuperscript{14}This target is used by dpkg\_buildpackage -B as in Section 6.2.

\textsuperscript{15}This target is used by dpkg\_buildpackage -A.

\textsuperscript{16}This uses the new debhelper v7+ features. Its design concepts are explained in Not Your Grandpa’s Debhelper (http://joey.kitenet.net/talks/debhelper-\textunderscore debhelper\_slides.pdf) presented at DebConf9 by the debhelper upstream. Under lenny, \texttt{dh\_make} created a much more complicated rules file with explicit rules and many \texttt{dh\_} scripts listed for each one, most of which are now unnecessary (and show the package’s age). The new \texttt{dh} command is simpler and frees us from doing the routine work “manually”. You still have full power to customize the process with \texttt{override\_dh\_} targets. See Section 4.4.3. It is based only on the debhelper package and does not obfuscate the package building process as the cdbs package tends to do.

\textsuperscript{17}You can verify the actual sequences of \texttt{dh\_} programs invoked for a given \texttt{target} without really running them by invoking \texttt{dh target --no-act} or debian/rules -- 'target --no-act'.

---

\textsuperscript{14}This target is used by dpkg\_buildpackage -B as in Section 6.2.

\textsuperscript{15}This target is used by dpkg\_buildpackage -A.

\textsuperscript{16}This uses the new debhelper v7+ features. Its design concepts are explained in Not Your Grandpa’s Debhelper (http://joey.kitenet.net/talks/debhelper-\textunderscore debhelper\_slides.pdf) presented at DebConf9 by the debhelper upstream. Under lenny, \texttt{dh\_make} created a much more complicated rules file with explicit rules and many \texttt{dh\_} scripts listed for each one, most of which are now unnecessary (and show the package’s age). The new \texttt{dh} command is simpler and frees us from doing the routine work “manually”. You still have full power to customize the process with \texttt{override\_dh\_} targets. See Section 4.4.3. It is based only on the debhelper package and does not obfuscate the package building process as the cdbs package tends to do.

\textsuperscript{17}You can verify the actual sequences of \texttt{dh\_} programs invoked for a given \texttt{target} without really running them by invoking \texttt{dh target --no-act} or debian/rules -- 'target --no-act'.

---
• **debian/rules build** runs **dh build**, which in turn runs the following:

  - `dh_testdir`
  - `dh_auto_configure`
  - `dh_auto_build`
  - `dh_auto_test`

• **fakeroot debian/rules binary** runs **fakeroot dh binary**, which in turn runs the following:\n
  - `dh_testroot`
  - `dh_prep`
  - `dh_installdirs`
  - `dh_auto_install`
  - `dh_install`
  - `dh_installdocs`
  - `dh_installchangelogs`
  - `dh_installexamples`
  - `dh_installman`
  - `dh_installcatalogs`
  - `dh_installcron`
  - `dh_installdebcconf`
  - `dh_installmacsen`
  - `dh_installfupdown`
  - `dh_installinfo`
  - `dh_installinit`
  - `dh_installmenu`
  - `dh_installmimewm`
  - `dh_installmodules`
  - `dh_installlogcheck`
  - `dh_installlogrotate`
  - `dh_installpam`
  - `dh_installpdp`
  - `dh_installudev`
  - `dh_installwm`
  - `dh_installxfonts`
  - `dh_bugfiles`
  - `dh_lintian`
  - `dh_gconf`
  - `dh_icons`
  - `dh_perl`
  - `dh_usrlocal`
  - `dh_link`
  - `dh_compress`
  - `dh_fixperms`
  - `dh_strip`
  - `dh_makeshlibs`
  - `dh_shlibdeps`
  - `dh_installdeb`
  - `dh_gencontrol`
  - `dh_md5sums`
  - `dh_builddeb`

• **fakeroot debian/rules binary-arch** runs **fakeroot dh binary-arch**, which in turn runs the same sequence as **fakeroot dh binary** but with the `-a` option appended for each command.

• **fakeroot debian/rules binary-indep** runs **fakeroot dh binary-indep**, which in turn runs almost the same sequence as **fakeroot dh binary** but excluding **`dh_strip`, `dh_makeshlibs`, and `dh_shlibdeps`** with the `-i` option appended for each remaining command.

---

\[\text{18}^{18}\text{The following example assumes your `debian/compat` has a value equal or more than 9 to avoid invoking any python support commands automatically.}\]
The functions of `dh_*` commands are largely self-evident from their names. There are a few notable ones that are worth giving (over)simplified explanations here assuming a typical build environment based on a `Makefile:

- **`dh_auto_clean`** usually executes the following if a `Makefile` exists with the `distclean` target.
  ```
  make distclean
  ```

- **`dh_auto_configure`** usually executes the following if `./configure` exists (arguments abbreviated for readability).
  ```
  ./configure --prefix=/usr --sysconfdir=/etc --localstatedir=/var ...
  ```

- **`dh_auto_build`** usually executes the following to execute the first target of `Makefile` if it exists.
  ```
  make
  ```

- **`dh_auto_test`** usually executes the following if a `Makefile` exists with the `test` target.
  ```
  make test
  ```

- **`dh_auto_install`** usually executes the following if a `Makefile` exists with the `install` target (line folded for readability).
  ```
  make install \
  DESTDIR=/path/to/package_version-revision/debian/package
  ```

All targets which require the `fakeroot` command will contain `dh_testroot`, which exits with an error if you are not using this command to pretend to be root.

The important part to know about the `rules` file created by `dh_make` is that it is just a suggestion. It will work for most packages but for more complicated ones, don’t be afraid to customize it to fit your needs.

Although `install` is not a required target, it is supported. `fakeroot dh install` behaves like `fakeroot dh binary` but stops after `dh_fixperms`.

### 4.4.3 Customization of rules file

There are many ways to customize the `rules` file created with the new `dh` command.

The `dh $@` command can be customized as follows:

- Add support for the `dh_python2` command. (The best choice for Python.)
  ```
  – Include the python package in `Build-Depends`.
  – Use `dh $@ --with python2`.
  – This handles Python modules using the `python` framework.
  ```

- Add support for the `dh_pysupport` command. (deprecated)
  ```
  – Include the `python-support` package in `Build-Depends`.
  – Use `dh $@ --with pysupport`.
  ```

---

19 For complete information on what all these `dh_*` scripts do exactly, and what their other options are, please read their respective manual pages and the `debhelper` documentation.

20 These commands support other build environments, such as `setup.py`, which can be listed by executing `dh_auto_build --list` in a package source directory.

21 It actually looks for the first available target in the `Makefile` out of `distclean`, `realclean`, or `clean`, and executes that.

22 If you install a package, you should activate its customization function by `dh $@ --with custom-name`.

23 Use of the `dh_python2` command is preferred over use of `dh_pysupport` or `dh_pycentral` commands. Do not use the `dh_python` command.
This handles Python modules using the python-support framework.

• Add support for the `dh_pycentral` command. (deprecated)
  – Include the python-central package in Build-Depends.
  – Use `dh $@ --with python-central` instead.
  – This also deactivates the `dh_pysupport` command.
  – This handles Python modules using the python-central framework.

• Add support for the `dh_installtex` command.
  – Include the tex-common package in Build-Depends.
  – Use `dh $@ --with tex` instead.
  – This registers Type 1 fonts, hyphenation patterns, and formats with TeX.

• Add support for the `dh_quilt_patch` and `dh_quilt_unpatch` commands.
  – Include the quilt package in Build-Depends.
  – Use `dh $@ --with quilt` instead.
  – This applies and un-applies patches to the upstream source from files in the debian/patches directory for a source package in the 1.0 format.
  – This is not needed if you use the new 3.0 (quilt) source package format.

• Add support for the `dh_dkms` command.
  – Include the dkms package in Build-Depends.
  – Use `dh $@ --with dkms` instead.
  – This correctly handles DKMS usage by kernel module packages.

• Add support for the `dh_autotools-dev_updateconfig` and `dh_autotools-dev_restoreconfig` commands.
  – Include the autotools-dev package in Build-Depends.
  – Use `dh $@ --with autotools-dev` instead.
  – This updates and restores config.sub and config.guess.

• Add support for the `dh_autoreconf` and `dh_autoreconf_clean` commands.
  – Include the dh-autoreconf package in Build-Depends.
  – Use `dh $@ --with autoreconf` instead.
  – This updates the GNU Build System files and restores them after the build.

• Add support for the `dh_girepository` command.
  – Includes the gobject-introspection package in Build-Depends.
  – Use `dh $@ --with gir` instead.
  – This computes dependencies for packages shipping GObject introspection data and generates the `${gir:Depends}` substitution variable for the package dependency.

• Add support for the `bash` completion feature.
  – Includes the bash-completion package in Build-Depends.
  – Use `dh $@ --with bash-completion` instead.
  – This installs `bash` completions using a configuration file at debian/package.bash-completion.
Many `dh_*` commands invoked by the new `dh` command can be customized by the corresponding configuration files in the `debian` directory. See Chapter 5 and the manpage of each command for the customization of such features.

You may need to run `dh_*` commands invoked via the new `dh` with added arguments, or to run additional commands with them, or to skip them. For such cases, you create an `override_dh_foo` target with its rule in the `rules` file defining an `override_dh_foo` target for the `dh_foo` command you want to change. It basically says *run me instead.*

Please note that the `dh_auto_*` commands tend to do more than what has been discussed in this (over)simplified explanation to take care of all the corner cases. It is a bad idea to use `override_dh_*` targets to substitute simplified equivalent commands (except for the `override_dh_auto_clean` target) since it may bypass such smart `debhelper` features.

So, for instance, if you want to store system configuration data in the `/etc/gentoo` directory instead of the usual `/etc` directory for the recent `gentoo` package using Autotools, you can override the default `--sysconfig=/etc` argument given by the `dh_auto_configure` command to the `.configure` command by the following:

```
override_dh_auto_configure:
  dh_auto_configure -- --sysconfig=/etc/gentoo
```

The arguments given after `--` are appended to the default arguments of the auto-executed program to override them. Using the `dh_auto_configure` command is better than directly invoking the `.configure` command here since it will only override the `--sysconfig` argument and retain any other, benign arguments to the `.configure` command.

If the `Makefile` in the source for `gentoo` requires you to specify `build` as its target to build it, you create an `override_dh_auto_build` target to enable this.

```
override_dh_auto_build:
  dh_auto_build -- build
```

This ensures `$(MAKE)` is run with all the default arguments given by the `dh_auto_build` command plus the `build` argument.

If the `Makefile` in the source for `gentoo` requires you to specify the `packageclean` target to clean it for the Debian package instead of using `distclean` or `clean` targets, you can create an `override_dh_auto_clean` target to enable it.

```
override_dh_auto_clean:
  $(MAKE) packageclean
```

If the `Makefile` in the source for `gentoo` contains a `test` target which you do not want to run for the Debian package building process, you can use an empty `override_dh_auto_test` target to skip it.

```
override_dh_auto_test:
```

If `gentoo` has an unusual upstream changelog file called `FIXES`, `dh_installchangelogs` will not install that file by default. The `dh_installchangelogs` command requires `FIXES` as its argument to install it.

```
override_dh_installchangelogs:
  dh_installchangelogs FIXES
```

When you use the new `dh` command, use of explicit targets such as the ones listed in Section 4.4.1, other than the `get-orig-source` target, may make it difficult to understand their exact effects. Please limit explicit targets to `override_dh_*` targets and completely independent ones, if possible.

---

25 Under `lenny`, if you wanted to change the behavior of a `dh_*` script you found the relevant line in the `rules` file and adjusted it.

26 `dh_auto_build` without any arguments will execute the first target in the `Makefile`.

27 The `debian/Changelog` and `debian/NEWS` files are always automatically installed. The upstream changelog is found by converting filenames to lower case and matching them against `changelog`, `changes`, `changelog.txt`, and `changes.txt`. 
Chapter 5

Other files under the debian directory

To control most of what debhelper does while building the package, you put optional configuration files under the debian directory. This chapter will provide an overview of what each of these does and its format. Please read the Debian Policy Manual (http://www.debian.org/doc/devel-manuals#policy) and Debian Developer’s Reference (http://www.debian.org/doc/devel-manuals#devref) for guidelines for packaging.

The dh_make command will create some template configuration files under the debian directory. Most of them come with filenames suffixed by .ex. Some of them come with filenames prefixed by the binary package name such as package. Take a look at all of them. \(^1\)

Some template configuration files for debhelper may not be created by the dh_make command. In such cases, you need to create them with an editor.

If you wish or need to activate any of these, please do the following:

- rename template files by removing the .ex or .EX suffix if they have one;
- rename the configuration files to use the actual binary package name in place of package;
- modify template file contents to suit your needs;
- remove template files which you do not need;
- modify the control file (see Section 4.1), if necessary;
- modify the rules file (see Section 4.4), if necessary.

Any debhelper configuration files without a package prefix, such as install, apply to the first binary package. When there are many binary packages, their configurations can be specified by prefixing their name to their configuration filenames such as package-1.install, package-2.install, etc.

5.1 README.Debian

Any extra details or discrepancies between the original package and your Debian version should be documented here.

dh_make created a default one; this is what it looks like:

```
gentoo for Debian
-------------------
<possible notes regarding this package - if none, delete this file>
-- Josip Rodin <joy-mg@debian.org>, Wed, 11 Nov 1998 21:02:14 +0100
```

If you have nothing to be documented, remove this file. See dh_installdocs(1).

\(^1\)In this chapter, files in the debian directory are referred to without the leading debian/ for simplicity whenever the meaning is obvious.
5.2 compat

The compat file defines the debhelper compatibility level. Currently, you should set it to the debhelper v10 as follows:

```
$ echo 10 > debian/compat
```

You may use compat level v9 in certain circumstances for compatibility with older systems. However, using any level below v9 is not recommended and should be avoided for new packages.

5.3 conffiles

One of the most annoying things about software is when you spend a great deal of time and effort customizing a program, only to have an upgrade stomp all over your changes. Debian solves this problem by marking such configuration files as conffiles. When you upgrade a package, you’ll be asked whether you want to keep your old configuration files or not.

dh_installdeb(1) automatically flags any files under the /etc directory as conffiles, so if your program only has conffiles there you do not need to specify them in this file. For most package types, the only place conffiles should ever be is under /etc, and so this file doesn’t need to exist.

If your program uses configuration files but also rewrites them on its own, it’s best not to make them conffiles because dpkg will then prompt users to verify the changes all the time.

If the program you’re packaging requires every user to modify the configuration files in the /etc directory, there are two popular ways to arrange for them to not be conffiles, keeping dpkg quiet:

- Create a symlink under the /etc directory pointing to a file under the /var directory generated by the maintainer scripts.
- Create a file generated by the maintainer scripts under the /etc directory.

For information on maintainer scripts, see Section 5.18.

5.4 package.cron.*

If your package requires regularly scheduled tasks to operate properly, you can use these files to set that up. You can set up regular tasks that either happen hourly, daily, weekly, or monthly, or alternatively happen at any other time that you wish. The filenames are:

- `package.cron.hourly` - Installed as /etc/cron.hourly/package; run once an hour.
- `package.cron.daily` - Installed as /etc/cron.daily/package; run once a day.
- `package.cron.weekly` - Installed as /etc/cron.weekly/package; run once a week.
- `package.cron.monthly` - Installed as /etc/cron.monthly/package: run once a month.
- `package.cron.d` - Installed as /etc/cron.d/package: for any other time.

Most of these files are shell scripts, with the exception of `package.cron.d` which follows the format of crontab(5).

No explicit cron.* file is needed to set up log rotation; for that, see dh_installlogrotate(1) and logrotate(8).

---

5.5 dirs

This file specifies any directories which we need but which are not created by the normal installation procedure (make install DESTDIR=... invoked by dh_auto_install). This generally means there is a problem with the Makefile.

Files listed in an install file don’t need their directories created first. See Section 5.11.

It is best to try to run the installation first and only use this if you run into trouble. There is no preceding slash on the directory names listed in the dirs file.

5.6 package.doc-base

If your package has documentation other than manual and info pages, you should use the doc-base file to register it, so the user can find it with e.g. dhelp(1), dwww(1), or doccentral(1).

This usually includes HTML, PS and PDF files, shipped in /usr/share/doc/packagename/.

This is what gentoo’s doc-base file gentoo.doc-base looks like:

| Document: | gentoo |
| Title: | Gentoo Manual |
| Author: | Emil Brink |
| Abstract: | This manual describes what Gentoo is, and how it can be used. |
| Section: | File Management |
| Format: | HTML |
| Files: | /usr/share/doc/gentoo/html/*.html |

For information on the file format, see install-docs(8) and the Debian doc-base manual at the local copy /usr/share/doc/doc-base/doc-base.html/index.html provided by the doc-base package.

For more details on installing additional documentation, look in Section 3.3.

5.7 docs

This file specifies the file names of documentation files we can have dh_installdocs(1) install into the temporary directory for us.

By default, it will include all existing files in the top-level source directory that are called BUGS, README*, TODO etc.

For gentoo, some other files are also included:

BUGS
CONFIG.CHANGES
CREDITS
NEWS
README
README.gtkrc
TODO

5.8 emacsen-*

If your package supplies Emacs files that can be bytecompiled at package installation time, you can use these files to set it up.

They are installed into the temporary directory by dh_installlemacsen(1).

If you don’t need these, remove them.
5.9  package.examples

The dh_installexamples(1) command installs files and directories listed in this file as example files.

5.10  package.init and package.default

If your package is a daemon that needs to be run at system start-up, you’ve obviously disregarded my initial recommendation, haven’t you? :-)

The package.init file is installed as the /etc/init.d/package script which starts and stops the daemon. Its fairly generic skeleton template is provided by the dh_make command as init.d.ex. You’ll likely have to rename and edit it, a lot, while making sure to provide Linux Standard Base (http://www.linuxfoundation.org/collaborate/workgroups/lsb) (LSB) compliant headers. It gets installed into the temporary directory by dh_installinit(1).

The package.default file will be installed as /etc/default/package. This file sets defaults that are sourced by the init script. This package.default file is most often used to set some default flags or timeouts. If your init script has certain configurable features, you can set them in the package.default file, instead of in the init script itself.

If your upstream program provides a file for the init script, you can either use it or not. If you don’t use their init script then create a new one in package.init. However if the upstream init script looks fine and installs in the right place you still need to set up the rc* symlinks. To do this you will need to override dh_installinit in the rules file with the following lines:

```
override_dh_installinit:
  dh_installinit --onlyscripts
```

If you don’t need this, remove the files.

5.11  install

If there are files that need to be installed into your package but your standard make install won’t do it, put the filenames and destinations into this install file. They are installed by dh_install(1). You should first check that there is not a more specific tool to use. For example, documents should be in the docs file and not in this one.

This install file has one line per file installed, with the name of the file (relative to the top build directory) then a space then the installation directory (relative to the install directory). One example of where this is used is if a binary src/bar is left unstalled; the install file might look like:

```
src/bar  usr/bin
```

This means when this package is installed, there will be an executable command /usr/bin/bar.

Alternatively, this install can have the name of the file only without the installation directory when the relative directory path does not change. This format is usually used for a large package that splits the output of its build into multiple binary packages using package-1.install, package-2.install, etc.

The dh_install command will fall back to looking in debian/tmp for files, if it doesn’t find them in the current directory (or wherever you’ve told it to look using --sourcedir).

5.12  package.info

If your package has info pages, you should install them using dh_installinfo(1) by listing them in a package.info file.

---

3This replaces the deprecated dh_movefiles(1) command which is configured by the files file.
5.13 **package.links**

If you need to create additional symlinks in package build directories as package maintainer, you should install them using `dh_link(1)` by listing their full paths of source and destination files in a *package*.links file.

5.14 {package,.source/}lintian-overrides

If lintian reports an erroneous diagnostic for a case where Debian policy allows exceptions to some rule, you can use *package*.lintian-overrides or *source*/lintian-overrides to quieten it. Please read the Lintian User’s Manual (https://lintian.debian.org/manual/index.html) and refrain from abusing this.

*package*.lintian-overrides is for the binary package named *package* and is installed into `usr/share/lintian/overrides/package` by the `dh_lintian` command.

*source*/lintian-overrides is for the source package. This is not installed.

5.15 **manpage.**

Your program(s) should have a manual page. If they don’t, you should create them. The `dh_make` command creates some template files for manual pages. These need to be copied and edited for each command missing its manual page. Please make sure to remove unused templates.

5.15.1 **manpage.1.ex**

Manual pages are normally written in nroff(1). The *manpage.1.ex* template is written in nroff, too. See the man(7) manual page for a brief description of how to edit such a file.

The final manual page file name should give the name of the program it is documenting, so we will rename it from *manpage* to *gentoo*. The file name also includes .1 as the first suffix, which means it’s a manual page for a user command. Be sure to verify that this section is indeed the correct one. Here’s a short list of manual page sections:

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>User commands</td>
<td>Executable commands or scripts</td>
</tr>
<tr>
<td>2</td>
<td>System calls</td>
<td>Functions provided by the kernel</td>
</tr>
<tr>
<td>3</td>
<td>Library calls</td>
<td>Functions within system libraries</td>
</tr>
<tr>
<td>4</td>
<td>Special files</td>
<td>Usually found in /dev</td>
</tr>
<tr>
<td>5</td>
<td>File formats</td>
<td>E.g. /etc/passwd’s format</td>
</tr>
<tr>
<td>6</td>
<td>Games</td>
<td>Games or other frivolous programs</td>
</tr>
<tr>
<td>7</td>
<td>Macro packages</td>
<td>Such as man macros</td>
</tr>
<tr>
<td>8</td>
<td>System administration</td>
<td>Programs typically only run by root</td>
</tr>
<tr>
<td>9</td>
<td>Kernel routines</td>
<td>Non-standard calls and internals</td>
</tr>
</tbody>
</table>

So *gentoo*’s man page should be called *gentoo.1*. If there was no *gentoo.1* man page in the original source, you should create it by renaming the *manpage.1.ex* template to *gentoo.1* and editing it using information from the example and from the upstream docs.

You can use the `help2man` command to generate a man page out of the --help and --version output of each program, too.

---

4Note that `help2man`’s placeholder man page will claim that more detailed documentation is available in the info system. If the command is missing an info page, you should manually edit the man page created by the `help2man` command.
5.15.2 manpage.sgml.ex

If on the other hand you prefer writing SGML instead of nroff, you can use the manpage.sgml.ex template. If you do this, you have to:

- rename the file to something like gentoo.sgml.
- install the docbook-to-man package
- add docbook-to-man to the Build-Depends line in the control file
- add an override_dh_auto_build target to your rules file:

```bash
override_dh_auto_build:
  docbook-to-man debian/gentoo.sgml > debian/gentoo.1
dh_auto_build
```

5.15.3 manpage.xml.ex

If you prefer XML over SGML, you can use the manpage.xml.ex template. If you do this, you have to:

- rename the source file to something like gentoo.1.xml
- install the docbook-xsl package and an XSLT processor like xsltproc (recommended)
- add the docbook-xsl, docbook-xml, and xsltproc packages to the Build-Depends line in the control file
- add an override_dh_auto_build target to your rules file:

```bash
override_dh_auto_build:
  xsltproc --nonet \ 
  --param make.year.ranges 1 \ 
  --param make.single.year.ranges 1 \ 
  --param man.charmap.use.subset 0 \ 
  -o debian/ \ 
  http://docbook.sourceforge.net/release/xsl/current/manpages/docbook.xsl\ 
  debian/gentoo.1.xml
dh_auto_build
```

5.16 package.manpages

If your package has manual pages, you should install them using dh_installman(1) by listing them in a package.manpages file.

To install docs/gentoo.1 as a manpage for the gentoo package, create a gentoo.manpages file as follows:

docs/gentoo.1

5.17 NEWS

The dh_installchangelogs(1) command installs this.
5.18 \{pre,post\}{inst,rm}

These postinst, preinst, postrm, and prerm files \(^5\) are called **maintainer scripts**. They are scripts which are put in the control area of the package and run by \texttt{dpkg} when your package is installed, upgraded, or removed.

As a novice maintainer, you should avoid any manual editing of maintainer scripts because they are problematic. For more information refer to the Debian Policy Manual, 6 “Package maintainer scripts and installation procedure” (http://www.debian.org/doc/debian-policy/ch-maintainerscripts.html), and take a look at the example files provided by \texttt{dh_make}.

If you did not listen to me and have created custom maintainer scripts for a package, you should make sure to test them not only for \texttt{install} and \texttt{upgrade} but also for \texttt{remove} and \texttt{purge}.

Upgrades to the new version should be silent and non-intrusive (existing users should not notice the upgrade except by discovering that old bugs have been fixed and perhaps that there are new features).

When the upgrade is necessarily intrusive (eg., config files scattered through various home directories with totally different structure), you may consider as the last resort switching the package to a safe fallback state (e.g., disabling a service) and providing the proper documentation required by policy (\texttt{README.Debian} and \texttt{NEWS.Debian}). Don’t bother the user with \texttt{debconf} notes invoked from these maintainer scripts for upgrades.

The \texttt{ucf} package provides a **conffile-like** handling infrastructure to preserve user changes for files that may not be labeled as \texttt{conf} files such as those managed by the maintainer scripts. This should minimize issues associated with them.

These maintainer scripts are among the Debian enhancements that explain why people choose Debian. You must be very careful not to turn them into a source of annoyance.

5.19 package.symbols

Packaging of a library is not easy for a novice maintainer and should be avoided. Having said it, if your package has libraries, you should have \texttt{debian/package.symbols} files. See Section A.2.

5.20 TODO

The \texttt{dh_installdocs(1)} command installs this.

5.21 watch

The \texttt{watch} file format is documented in the \texttt{uscan(1)} manpage. The \texttt{watch} file configures the \texttt{uscan} program (in the \texttt{devscripts} package) to watch the site where you originally got the source. This is also used by the Debian Package Tracker (https://tracker.debian.org/) service.

Here are its contents:

```bash
# watch control file for uscan
version=3
http://sf.net/gentoo/gentoo-(.+)
.tar.gz debian uupdate
```

Normally with a \texttt{watch} file, the URL at \texttt{http://sf.net/gentoo} is downloaded and searched for links of the form \texttt{<a href=...>}. The basename (just the part after the final /) of each linked URL is compared against the Perl regular expression pattern (see \texttt{perlr(1)}) \texttt{gentoo-(.+)
.tar.gz}. Out of the files that match, the one with the greatest version number is downloaded and the \texttt{uupdate} program is run to create an updated source tree.

\(^5\)Despite this use of the \texttt{bash} shorthand expression \{pre,post\}{inst,rm} to indicate these filenames, you should use pure POSIX syntax for these maintainer scripts for compatibility with \texttt{dash} as the system shell.
Although this is true for other sites, the SourceForge download service at [http://sf.net](http://sf.net) is an exception. When the watch file has a URL matching the Perl regexp `^http://sf\.net/`, the `uscan` program replaces it with `http://qa.debian.org/watch/sf.php` and then applies this rule. The URL redirector service at [http://qa.debian.org/](http://qa.debian.org/) is designed to offer a stable redirect service to the desired file for any watch pattern of the form `http://sf.net/project/tar-name-(.+)/tar.gz`. This solves issues related to periodically changing SourceForge URLs.

If the upstream offers the cryptographic signature of the tarball, it is recommended to verify its authenticity using the `pgpsigurlmangle` option as described in `uscan(1)`.

## 5.22 source/format

In the `debian/source/format` file, there should be a single line indicating the desired format for the source package (check `dpkg-source(1)` for an exhaustive list). After `squeeze`, it should say either:

- 3.0 (native) for native Debian packages or
- 3.0 (quilt) for everything else.

The newer 3.0 (quilt) source format records modifications in a quilt patch series within `debian/patches`. Those changes are then automatically applied during extraction of the source package. The Debian modifications are simply stored in a `debian.tar.gz` archive containing all files under the `debian` directory. This new format supports inclusion of binary files such as PNG icons by the package maintainer without requiring tricks.

When `dpkg-source` extracts a source package in 3.0 (quilt) source format, it automatically applies all patches listed in `debian/patches/series`. You can avoid applying patches at the end of the extraction with the `--skip-patches` option.

## 5.23 source/local-options

When you want to manage Debian packaging activities under a VCS, you typically create one branch (e.g., `upstream`) tracking the upstream source and another branch (e.g., typically `master` for Git) tracking the Debian package. For the latter, you usually want to have unpatched upstream source with your `debian/*` files for the Debian packaging to ease merging of the new upstream source.

After you build a package, the source is normally left patched. You need to unpatch it manually by running `dquilt pop -a` before committing to the `master` branch. You can automate this by adding the optional `debian/source/local-options` file containing `unapply-patches`. This file is not included in the generated source package and changes the local build behavior only. This file may contain `abort-on-upstream-changes`, too (see `dpkg-source(1)`).

```
unapply-patches
abort-on-upstream-changes
```

## 5.24 source/options

The autogenerated files in the source tree can be quite annoying for packaging since they generate meaningless large patch files. There are custom modules such as `dh_autoreconf` to ease this problem as described in Section 4.4.3.

You can provide a Perl regular expression to the `--extend-diff-ignore` option argument of `dpkg-source(1)` to ignore changes made to the autogenerated files while creating the source package.

As a general solution to address this problem of the autogenerated files, you can store such a `dpkg-source` option argument in the `source/options` file of the source package. The following will skip creating patch files for `config.sub`, `config.guess`, and `Makefile`.

```
extend-diff-ignore = "(^|/)\(config\sub|config\guess|Makefile\)$"
```

---

6See [DebSrc3.0](http://wiki.debian.org/Projects/DebSrc3.0) for a summary on the switch to the new 3.0 (quilt) and 3.0 (native) source formats.

7Actually, this new format also supports multiple upstream tarballs and more compression methods. These are beyond the scope of this document.
5.25 patches/*

The old 1.0 source format created a single large `diff.gz` file containing package maintenance files in `debian` and patch files for the source. Such a package is a bit cumbersome to inspect and understand for each source tree modification later. This is not so nice.

The newer 3.0 (quilt) source format stores patches in `debian/patches/*` files using the `quilt` command. These patches and other package data which are all contained under the `debian` directory are packaged as the `debian.tar.gz` file. Since the `dpkg-source` command can handle `quilt` formatted patch data in the 3.0 (quilt) source without the `quilt` package, it does not need a `Build-Depends` on `quilt`.

The `quilt` command is explained in `quilt(1)`. It records modifications to the source as a stack of `-p1` patch files in the `debian/patches` directory and the source tree is untouched outside of the `debian` directory. The order of these patches is recorded in the `debian/patches/series` file. You can apply (=push), un-apply (=pop), and refresh patches easily.

For Chapter 3, we created three patches in `debian/patches`.

Since Debian patches are located in `debian/patches`, please make sure to set up the `dquilt` command properly as described in Section 3.1.

When anyone (including yourself) provides a patch `foo.patch` to the source later, modifying a 3.0 (quilt) source package is quite simple:

```
$ dpkg-source -x gentoo_0.9.12.dsc
$ cd gentoo-0.9.12
$ dquilt import ../foo.patch
$ dquilt push
$ dquilt refresh
$ dquilt header -e
... describe patch
```

The patches stored in the newer 3.0 (quilt) source format must be `fuzz free`. You can ensure this with `dquilt pop -a; while dquilt push; do dquilt refresh; done`.

---

8Several methods of patch set maintenance have been proposed and are in use for Debian packages. The `quilt` system is the preferred maintenance system in use. Others include `dpatch`, `dbs`, and `cibs`. Many of these keep such patches as `debian/patches/*` files.

9If you are asking a sponsor to upload your package, this kind of clear separation and documentation of your changes is very important to expedite the package review by your sponsor.
Chapter 6

Building the package

We should now be ready to build the package.

6.1 Complete (re)build

In order to perform a complete (re)build of a package properly, you need to make sure you have installed

- the build-essential package,
- packages listed in the Build-Depends field (see Section 4.1), and
- packages listed in the Build-Depends-indep field (see Section 4.1).

Then you issue the following command in the source directory:

```
$ dpkg-buildpackage -us -uc
```

This will do everything to make full binary and source packages for you. It will:

- clean the source tree (debian/rules clean)
- build the source package (dpkg-source -b)
- build the program (debian/rules build)
- build binary packages (fakeroot debian/rules binary)
- make the .dsc file
- make the .changes file, using dpkg-genchanges

If the build result is satisfactory, sign the .dsc and .changes files with your private GPG key using the debsign command. You need to enter your secret pass phrase, twice. ¹

For a non-native Debian package, e.g., gentoo, you will see the following files in the parent directory (~/gentoo) after building packages:

¹This GPG key must be signed by a Debian developer to get connected to the web of trust and must be registered to the Debian keyring (http://keyring.debian.org). This enables your uploaded packages to be accepted to the Debian archives. See Creating a new GPG key (http://keyring.debian.org/-creating-key.html) and Debian Wiki on Keysigning (http://wiki.debian.org/Keysigning).
• gentoo_0.9.12.orig.tar.gz
  This is the original upstream source code tarball, merely renamed to the above so that it adheres to the Debian standard. Note that this was created initially by the command dh_make -f ../gentoo-0.9.12.tar.gz.

• gentoo_0.9.12-1.dsc
  This is a summary of the contents of the source code. The file is generated from your control file, and is used when unpacking the source with dpkg-source(1).

• gentoo_0.9.12-1.debian.tar.gz
  This compressed tarball contains your debian directory contents. Each and every addition you made to the original source code is stored as a quilt patch in debian/patches.
  If someone else wants to re-create your package from scratch, they can easily do so using the above three files. The extraction procedure is trivial: just copy the three files somewhere else and run dpkg-source -x gentoo_0.9.12-1.dsc.²

• gentoo_0.9.12-1_i386.deb
  This is your completed binary package. You can use dpkg to install and remove this just like any other package.

• gentoo_0.9.12-1_i386.changes
  This file describes all the changes made in the current package revision; it is used by the Debian FTP archive maintenance programs to install the binary and source packages. It is partly generated from the changelog file and the .dsc file.
  As you keep working on the package, its behavior will change and new features will be added. People downloading your package can look at this file and quickly see what has changed. Debian archive maintenance programs will also post the contents of this file to the debian-devel-changes@lists.debian.org (http://lists.debian.org/debian-devel-changes/) mailing list.

The gentoo_0.9.12-1.dsc and gentoo_0.9.12-1_i386.changes files must be signed using the debsign command with your private GPG key in the ~/.gnupg/ directory, before uploading them to the Debian FTP archive. The GPG signature provides the proof that these files are really yours, using your public GPG key.

The debsign command can be made to sign with your specified secret GPG key ID (good for sponsoring packages) with the following in the ~/.devscripts file:

DEBSIGN_KEYID=Your_GPG_keyID

The long strings of numbers in the .dsc and .changes files are SHA1/SHA256 checksums for the files mentioned. Anyone downloading your files can test them with sha1sum(1) or sha256sum(1) and if the numbers don’t match, they’ll know the file is corrupt or has been tampered with.

6.2 Autobuilder

Debian supports many ports (http://www.debian.org/ports/) with the autobuilder network (http://www.debian.org/devel/builddd/) running buildd daemons on computers of many different architectures. Although you do not need to do this yourself, you should be aware of what will happen to your packages. Let’s look into roughly how they rebuild your packages for multiple architectures.

For Architecture: any packages, the autobuilder system performs a rebuild. It ensures the installation of

• the build-essential package, and
• packages listed in the Build-Depends field (see Section 4.1).

Then it issues the following command in the source directory:

²You can avoid applying quilt patches in the 3.0 (quilt) source format at the end of the extraction with the --skip-patches option. Alternatively, you can run dquilt pop -a after normal operation.

³The actual autobuilder system involves much more complicated schemes than the one documented here. Such details are beyond the scope of this document.
$ dpkg-buildpackage -B

This will do everything to make architecture dependent binary packages on another architecture. It will:

- clean the source tree (debian/rules clean)
- build the program (debian/rules build)
- build architecture dependent binary packages (fakeroot debian/rules binary-arch)
- sign the source .dsc file, using gpg
- create and sign the upload .changes file, using dpkg-genchanges and gpg

This is why you see your package for other architectures.

Although packages listed in the Build-Depends-Indep field are required to be installed for our normal packaging work (see Section 6.1), they are not required to be installed for the autobuilder system since it builds only architecture dependent binary packages. ¹ This distinction between normal packaging and autobuilding procedures is what dictates whether you should record such required packages in the Build-Depends or Build-Depends-Indep fields of the debian/control file (see Section 4.1).

### 6.3 debuild command

You can automate the build activity around executing the dpkg-buildpackage command package further with the debuild command. See debuild(1).

The debuild command executes the lintian command to make a static check after building the Debian package. The lintian command can be customized with the following in the ~/.devscripts file:

```bash
DEBUILD_DPKG_BUILDPACKAGE_OPTS="-us -uc -I -i"
DEBUILD_LINTIAN_OPTS="-I --show-overrides"
```

Cleaning the source and rebuilding the package from your user account is as simple as:

```
$ debuild
```

You can clean the source tree as simply as:

```
$ debuild -- clean
```

### 6.4 pbuilder package

For a clean room (chroot) build environment to verify the build dependencies, the pbuilder package is very useful. ⁵ This ensures a clean build from the source under the sid auto-builder for different architectures and avoids a severity serious FTBFS (Fails To Build From Source) bug which is always in the RC (release critical) category. ⁶

Let’s customize the pbuilder package as follows:

- setting the /var/cache/pbuilder/result directory writable by your user account.

---

¹Unlike under the pbuilder package, the chroot environment under the sbuild package used by the autobuilder system does not enforce the use of a minimal system and may have many leftover packages installed.

⁵Since the pbuilder package is still evolving, you should check the actual configuration situation by consulting the latest official documentation.

⁶See http://buildd.debian.org/ for more on Debian package auto-building.
• creating a directory, e.g. `/var/cache/pbuilder/hooks`, writable by the user, to place hook scripts in.

• configuring `~/.pbuilderrc` or `/etc/pbuilderrc` to include the following.

```
AUTO_DEBSIGN=${AUTO_DEBSIGN:-no}
HOOKDIR=/var/cache/pbuilder/hooks
```

First let's initialize the local pbuilder chroot system as follows:

```
$ sudo pbuilder create
```

If you already have a completed source package, issue the following commands in the directory where the `foo.orig.tar.gz`, `foo.debian.tar.gz`, and `foo.dsc` files exist to update the local pbuilder chroot system and to build binary packages in it:

```
$ sudo pbuilder --update
$ sudo pbuilder --build foo_version.dsc
```

The newly built packages without the GPG signatures will be located in `/var/cache/pbuilder/result/` with non-root ownership.

The GPG signatures on the `.dsc` file and the `.changes` file can be generated as:

```
$ cd /var/cache/pbuilder/result/
$ debsign foo_version_arch.changes
```

If you have an updated source tree but have not generated the matching source package, issue the following commands in the source directory where the `debian` directory exists, instead:

```
$ sudo pbuilder --update
$ pdebuild
```

You can log into its chroot environment with the `pbuilder --login --save-after-login` command and configure it as you wish. This environment can be saved by leaving its shell prompt with ^D (Control-D).

The latest version of the lintian command can be executed in the chroot environment using the hook script `/var/cache/pbuilder/hooks/B90lintian` configured as follows:  

```sh
#!/bin/sh
set -e
install_packages() {
    apt-get -y --allow-downgrades install "$@
}
install_packages lintian
echo "+++ lintian output +++"
su -c "lintian -i -I --show-overrides /tmp/buildd/*/changes" - pbuilder
# use this version if you don't want lintian to fail the build
#su -c "lintian -i -I --show-overrides /tmp/buildd/*/changes; :" - pbuilder
echo "+++ end of lintian output +++"
```

You need to have access to the latest sid environment to build packages properly for sid. In practice, sid may be experiencing issues which makes it undesirable for you to migrate your whole system. The pbuilder package can help you to cope with this kind of situation.

You may need to update your stable packages after their release for `stable-proposed-updates`, `stable/updates`, etc.  

For such occasions, the fact that you may be running a sid system is not a good enough excuse for failing to update them promptly. The pbuilder package can help you to access environments of almost any Debian derivative distribution of the same architecture.


---

7This assumes `HOOKDIR=/var/cache/pbuilder/hooks`. You can find many examples of hook scripts in the `/usr/share/doc/pbuilder/examples` directory.

8There are some restrictions for such updates of your `stable` package.
6.5 git-buildpackage command and similar

If your upstream uses a source code management system (VCS)\(^9\) to maintain their code, you should consider using it as well. This makes merging and cherry-picking upstream patches much easier. There are several specialized wrapper script packages for Debian package building for each VCS.

- **git-buildpackage**: a suite to help with Debian packages in Git repositories.
- **svn-buildpackage**: helper programs to maintain Debian packages with Subversion.
- **cvs-buildpackage**: a set of Debian package scripts for CVS source trees.

Use of **git-buildpackage** is becoming quite popular for Debian Developers to manage Debian packages with the Git server on [alioth.debian.org](http://alioth.debian.org/).\(^{10}\) This package offers many commands to *automate* packaging activities:

- `gbp-import-dsc(1)`: import a previous Debian package to a Git repository.
- `gbp-import-orig(1)`: import a new upstream tar to a Git repository.
- `gbp-dch(1)`: generate the Debian changelog from Git commit messages.
- `git-buildpackage(1)`: build Debian packages from a Git repository.
- `git-pbuilder(1)`: build Debian packages from a Git repository using `pbuilder/cowbuilder`.

These commands use 3 branches to track packaging activity:

- **main** for Debian package source tree.
- **upstream** for upstream source tree.
- **pristine-tar** for upstream tarball generated by the `--pristine-tar` option.\(^{11}\)

You can configure **git-buildpackage** with `~/.gbp.conf`. See `gbp.conf(5)`.\(^{12}\)

6.6 Quick rebuild

With a large package, you may not want to rebuild from scratch every time while you’re tuning details in `debian/rules`. For testing purposes, you can make a `.deb` file without rebuilding the upstream sources like this\(^{13}\):

```
$ fakeroot debian/rules binary
```

Or simply do the following to see if it builds or not:

```
$ fakeroot debian/rules build
```

Once you are finished with your tuning, remember to rebuild following the proper procedure. You may not be able to upload correctly if you try to upload `.deb` files built this way.

---

\(^9\)See [Version control systems](http://www.debian.org/doc/manuals/debian-reference/ch10#_version_control_systems) for more.


\(^{11}\)The `--pristine-tar` option invokes the `pristine-tar` command, which can regenerate an exact copy of a pristine upstream tarball using only a small binary delta file and the contents of the tarball that are typically kept in an upstream branch in the VCS.

\(^{12}\)Here are some web resources available for advanced audiences.

- `debian packages in git` ([https://honk.sigxcpu.org/piki/development/debian_packages_in_git/](https://honk.sigxcpu.org/piki/development/debian_packages_in_git/))
- Using Git for Debian Packaging ([http://www.eagle.org/~eagle/notes/debian/git.html](http://www.eagle.org/~eagle/notes/debian/git.html))
- `git-dpm`: Debian packages in Git manager ([http://git-dpm.alioth.debian.org/](http://git-dpm.alioth.debian.org/))

\(^{13}\)Environment variables that are normally configured to proper values are not set by this method. Never create real packages to be uploaded using this `quick` method.
6.7 Command hierarchy

Here is a quick summary of how many commands to build packages fit together in the command hierarchy. There are many ways to do the same thing.

- `debian/rules` = maintainer script for the package building
- `dpkg-buildpackage` = core of the package building tool
- `debuild = dpkg-buildpackage + lintian` (build under the sanitized environment variables)
- `pbuilder` = core of the Debian chroot environment tool
- `pdebuild = pbuilder + dpkg-buildpackage` (build in the chroot)
- `cowbuilder` = speed up the `pbuilder` execution
- `git-pbuilder` = the easy-to-use commandline syntax for `pdebuild` (used by `gbp buildpackage`)
- `gbp` = manage the Debian source under the git repo
- `gbp buildpackage = pbuilder + dpkg-buildpackage + gbp`

Although use of higher level commands such as `gbp buildpackage` and `pbuilder` ensures the perfect package building environment, it is essential to understand how lower level commands such as `debian/rules` and `dpkg-buildpackage` are executed under them.
Chapter 7
Checking the package for errors

There are some techniques you should know for checking a package for errors before uploading it to the public archives.
It’s also a good idea to carry out testing on a machine other than your own. You must watch closely for any warnings or errors for all the tests described here.

7.1 Suspicious changes

If you find a new autogenerated patch file such as debian-changes-* in the debian/patches directory after building your non-native Debian package in 3.0 (quilt) format, chances are you changed some files by accident or the build script modified the upstream source. If it is your mistake, fix it. If it is caused by the build script, fix the root cause with dh-autoreconf as in Section 4.4.3 or workaround it with source/options as in Section 5.24.

7.2 Verifying a package’s installation

You must test your package for whether it installs without problems. The debi(1) command helps you to test installing all the generated binary packages.

$ sudo debi gentoo_0.9.12-1_i386.changes

To prevent installation problems on different systems, you must make sure that there are no filenames conflicting with other existing packages, using the Contents-i386 file downloaded from the Debian archive. The apt-file command may be handy for this task. If there are collisions, please take action to avoid this real problem, whether by renaming the file, moving a common file to a separate package that multiple packages can depend on, using the alternatives mechanism (see update-alternatives(1)) in coordination with the maintainers of other affected packages, or declaring a Conflicts relationship in the debian/control file.

7.3 Verifying a package’s maintainer scripts

All maintainer scripts (that is, preinst, prerm, postinst, and postrm files) are hard to write correctly unless they are auto-generated by the debhelper programs. So do not use them if you are a novice maintainer (see Section 5.18).

If the package makes use of these non-trivial maintainer scripts, be sure to test not only for install but also for remove, purge, and upgrade processes. Many maintainer script bugs show up when packages are removed or purged. Use the dpkg command as follows to test them:

$ sudo dpkg -r gentoo
$ sudo dpkg -P gentoo
$ sudo dpkg -i gentoo_version-revision_i386.deb
This should be done with sequences such as the following:

- install the previous version (if needed).
- upgrade it from the previous version.
- downgrade it back to the previous version (optional).
- purge it.
- install the new package.
- remove it.
- install it again.
- purge it.

If this is your first package, you should create dummy packages with different versions to test your package in advance to prevent future problems.

Bear in mind that if your package has previously been released in Debian, people will often be upgrading to your package from the version that was in the last Debian release. Remember to test upgrades from that version too.

Although downgrading is not officially supported, supporting it is a friendly gesture.

### 7.4 Using lintian

Run lintian(1) on your `.changes` file. The `lintian` command runs many test scripts to check for many common packaging errors.  

```
$ lintian -i -I --show-overrides gentoo_0.9.12-1_i386.changes
```

Of course, replace the filename with the name of the `.changes` file generated for your package. The output of the `lintian` command uses the following flags:

- **E**: for error; a sure policy violation or packaging error.
- **W**: for warning; a possible policy violation or packaging error.
- **I**: for info; information on certain aspects of packaging.
- **N**: for note; a detailed message to help your debugging.
- **O**: for overridden; a message overridden by the `lintian-overrides` files but displayed by the `--show-overrides` option.

When you see warnings, tune the package to avoid them or verify that the warnings are spurious. If spurious, set up `lintian-overrides` files as described in Section 5.14.

Note that you can build the package with `dpkg-buildpackage` and run `lintian` on it in one command, if you use `debuild(1)` or `pdebuild(1)`.

---

1You do not need to provide the `lintian` option `-i -I --show-overrides` if you customized `/etc/devscripts.conf` or `~/.devscripts` as described in Section 6.3.
7.5 The debc command

You can list files in the binary Debian package with the debc(1) command.

```
$ debc package.changes
```

7.6 The debdiff command

You can compare file contents in two source Debian packages with the debdiff(1) command.

```
$ debdiff old-package.dsc new-package.dsc
```

You can also compare file lists in two sets of binary Debian packages with the debdiff(1) command.

```
$ debdiff old-package.changes new-package.changes
```

These are useful to identify what has been changed in the source packages and to check for inadvertent changes made when updating binary packages, such as unintentionally misplacing or removing files.

7.7 The interdiff command

You can compare two diff.gz files with the interdiff(1) command. This is useful for verifying that no inadvertent changes were made to the source by the maintainer when updating packages in the old 1.0 source format.

```
$ interdiff -z old-package.diff.gz new-package.diff.gz
```

The new 3.0 source format stores changes in multiple patch files as described in Section 5.25. You can trace changes of each debian/patches/* file using interdiff, too.

7.8 The mc command

Many of these file inspection operations can be made into an intuitive process by using a file manager like mc(1) which will let you browse not only the contents of *.deb package files but also *.udeb, *.debian.tar.gz, *.diff.gz, and *.orig.tar.gz files.

Be on the lookout for extra unneeded files or zero length files, both in the binary and source package. Often cruft doesn’t get cleaned up properly; adjust your rules file to compensate for this.
Chapter 8

Updating the package

After you release a package, you will soon need to update it.

8.1 New Debian revision

Let’s say that a bug report was filed against your package as #654321, and it describes a problem that you can solve. Here’s what you need to do to create a new Debian revision of the package:

• If this is to be recorded as a new patch, do the following:
  – `dquilt new bugname.patch` to set the patch name;
  – `dquilt add buggy-file` to declare the file to be modified;
  – Correct the problem in the package source for the upstream bug;
  – `dquilt refresh` to record it to `bugname.patch`;
  – `dquilt header -e` to add its description;

• If this is to update an existing patch, do the following:
  – `dquilt pop foo.patch` to recall the existing `foo.patch`;
  – Correct the problem in the old `foo.patch`;
  – `dquilt refresh` to update `foo.patch`;
  – `dquilt header -e` to update its description;
  – `while dquilt push; do dquilt refresh; done` to apply all patches while removing `fuzz`;

• Add a new revision at the top of the Debian changelog file, for example with `dch -i`, or explicitly with `dch -v version-revision` and then insert the comments using your preferred editor. 1

• Include a short description of the bug and the solution in the changelog entry, followed by Closes: #654321. That way, the bug report will be automagically closed by the archive maintenance software the moment your package gets accepted into the Debian archive.

• Repeat what you did in the above to fix more bugs while updating the Debian changelog file with `dch` as needed.

• Repeat what you did in Section 6.1 and Chapter 7.

1To get the date in the required format, use LANG=C date -R.
• Once you are satisfied, you should change the distribution value in `changelog` from `UNRELEASED` to the target distribution value `unstable` (or even `experimental`).

• Upload the package as in Chapter 9. The difference is that this time, the original source archive won’t be included, as it hasn’t been changed and it already exists in the Debian archive.

One tricky case can occur when you make a local package, to experiment with the packaging before uploading the normal version to the official archive, e.g., `1.0.1-1`. For smoother upgrades, it is a good idea to create a `changelog` entry with a version string such as `1.0.1-1~rc1`. You may unclutter `changelog` by consolidating such local change entries into a single entry for the official package. See Section 2.6 for the order of version strings.

8.2 Inspection of the new upstream release

When preparing packages of a new upstream release for the Debian archive, you must check the new upstream release first.

Start by reading the upstream `changelog`, `NEWS`, and whatever other documentation they may have released with the new version.

You can then inspect changes between the old and new upstream sources as follows, watching out for anything suspicious:

```bash
$ diff -urN foo-oldversion foo-newversion
```

Changes to some auto-generated files by Autotools such as `missing`, `aclocal.m4`, `config.guess`, `config.h.in`, `config.sub`, `configure`, `depcomp`, `install-sh`, `ltmain.sh`, and `Makefile.in` may be ignored. You may delete them before running `diff` on the source for inspection.

8.3 New upstream release

If a package `foo` is properly packaged in the newer `3.0 (native)` or `3.0 (quilt)` formats, packaging a new upstream version is essentially moving the old `debian` directory to the new source. This can be done by running `tar xvf /path/to/foo_oldversion` in the new extracted source. Of course, you need to do some obvious chores:

• Create a copy of the upstream source as the `foo_newversion.orig.tar.gz` file.

• Update the Debian changelog file with `dch -v newversion-1`.

  – Add an entry with `New upstream release`.
  – Describe concisely the changes in the new upstream release that fix reported bugs and close those bugs by adding `Closes: #bug_number`.
  – Describe concisely the changes to the new upstream release by the maintainer that fix reported bugs and close those bugs by adding `Closes: #bug_number`.

• while `dquilt push`; do `dquilt refresh`; done to apply all patches while removing `fuzz`.

If the patch/merge did not apply cleanly, inspect the situation (clues are left in `.rej` files).

• If a patch you applied to the source was integrated into the upstream source,

  – `dquilt delete` to remove it.

• If a patch you applied to the source conflicted with new changes in the upstream source,

  2 If you use the `dch -r` command to make this last change, please make sure to save the `changelog` file explicitly by the editor.

  3 If a package `foo` is packaged in the old `1.0` format, this can be done by running `zcat /path/to/foo_oldversion.diff.gz|patch -p1` in the new extracted source, instead.
- `d quilt push -f` to apply old patches while forcing rejects as `baz.rej`.
- Edit the `baz` file manually to bring about the intended effect of `baz.rej`.
- `d quilt refresh` to update the patch.

• Continue as usual with `while d quilt push; do d quilt refresh; done`.

This process can be automated using the `uupdate(1)` command as follows:

```bash
$ apt-get source foo ...

dpkg-source: info: extracting foo in foo-oldversion
dpkg-source: info: unpacking foo_oldversion.orig.tar.gz
dpkg-source: info: applying foo_oldversion-1.debian.tar.gz
$ ls -F
foo_oldversion/
foo_oldversion-1.debian.tar.gz
foo_oldversion-1.dsc
foo_oldversion.orig.tar.gz
$ wget http://example.org/foo/foo-newversion.tar.gz
$ cd foo-oldversion
$ uupdate -v newversion ../foo-newversion.tar.gz
$ cd ../foo-newversion
$ while d quilt push; do d quilt refresh; done
$ dch ...
```

Document changes made

If you set up a `debian/watch` file as described in Section 5.21, you can skip the `wget` command. You simply run `uscan(1)` in the `foo-oldversion` directory instead of the `uupdate` command. This will automatically look for the updated source, download it, and run the `uupdate` command. 4

You can release this updated source by repeating what you did in Section 6.1, Chapter 7, and Chapter 9.

### 8.4 Updating the packaging style

Updating the package style is not a required activity for the update of a package. However, doing so lets you use the full capabilities of the modern `debhelper` system and the 3.0 source format. 5

• If you need to recreate deleted template files for any reason, you can run `dh_make` again in the same Debian package source tree with the `--addmissing` option. Then edit them appropriately.

• If the package has not been updated to use the `debhelper v7+ dh` syntax for the `debian/rules` file, update it to use `dh`. Update the `debian/control` file accordingly.

• If you want to update the `rules` file created with the `Makefile` inclusion mechanism of the Common Debian Build System (CDBS) to the `dh` syntax, see the following to understand its `DEB_*` configuration variables.

  - local copy of `/usr/share/doc/cdbs/cdbs-doc.pdf.gz`
  - The Common Debian Build System (CDBS), FOSDEM 2009 (http://meetings-archive.debian.net/pub/debian-meetings/-2009/fosdem/slides/The_Common_Debian_Build_System_CDBS/)

• If you have a 1.0 source package without the `foo.diff.gz` file, you can update it to the newer 3.0 (native) source format by creating `debian/source/format` with 3.0 (native). The rest of the `debian/*` files can just be copied.

---

4 If the `uscan` command downloads the updated source but it does not run the `uupdate` command, you should correct the `debian/watch` file to have `debian uupdate` at the end of the URL.

5 If your sponsor or other maintainers object to updating the existing packaging style, don’t bother arguing. There are more important things to do.
If you have a 1.0 source package with the *foo*.diff.gz file, you can update it to the newer 3.0 (quilt) source format by creating debian/source/format with 3.0 (quilt). The rest of the debian/* files can just be copied. Import the big.diff file generated by the command `filterdiff -z -x '/debian/\*' *foo*.diff.gz > big.diff` to your quilt system, if needed.

If it was packaged using another patch system such as dpatch, dbs, or cdbs with -p0, -p1, or -p2, convert it to quilt using deb3 at http://bugs.debian.org/581186.

If it was packaged with the dh command with the --with quilt option or with the dh_quilt_patch and dh_quilt_unpatch commands, remove these and make it use the newer 3.0 (quilt) source format.

You should check DEP - Debian Enhancement Proposals (http://dep.debian.net/) and adopt ACCEPTED proposals. You need to do the other tasks described in Section 8.3, too.

### 8.5 UTF-8 conversion

If upstream documents are encoded in old encoding schemes, converting them to UTF-8 is a good idea.

- Use `iconv(1)` to convert encodings of plain text files.
  ```
  iconv -f latin1 -t utf8 *foo_in.txt > *foo_out.txt
  ```

- Use `w3m(1)` to convert from HTML files to UTF-8 plain text files. When you do this, make sure to execute it under UTF-8 locale.
  ```
  LC_ALL=en_US.UTF-8 w3m -o display_charset=UTF-8 \\
     -cols 70 -dump -no-graph -T text/html \\
     < *foo_in.html > *foo_out.txt
  ```

### 8.6 Reminders for updating packages

Here are a few reminders for updating packages:

- Preserve old changelog entries (sounds obvious, but there have been cases of people typing dch when they should have typed dch -i.)

- Existing Debian changes need to be reevaluated; throw away stuff that upstream has incorporated (in one form or another) and remember to keep stuff that hasn’t been incorporated by upstream, unless there is a compelling reason not to.

- If any changes were made to the build system (hopefully you’d know from inspecting upstream changes) then update the debian/rules and debian/control build dependencies if necessary.

- Check the Debian Bug Tracking System (BTS) (http://www.debian.org/Bugs/) to see if someone has provided patches to bugs that are currently open.

- Check the contents of the .changes file to make sure you are uploading to the correct distribution, the proper bug closures are listed in the Closes field, the Maintainer and Changed-By fields match, the file is GPG-signed, etc.

---

6 You can split big.diff into many small incremental patches using the `splitdiff` command.
Chapter 9

Uploading the package

Now that you have tested your new package thoroughly, you want to release it to a public archive to share it.

9.1 Uploading to the Debian archive

Once you become an official developer, you can upload the package to the Debian archive. You can do this manually, but it’s easier to use the existing automated tools, like dupload(1) or dput(1). We’ll describe how it’s done with dupload.

First you have to set up dupload’s config file. You can either edit the system-wide /etc/dupload.conf file, or have your own ~/.dupload.conf file override the few things you want to change.

You can read the dupload.conf(5) manual page to understand what each of these options means.

The $default_host option determines which of the upload queues will be used by default. anonymous-ftp-master is the primary one, but it’s possible that you will want to use another one.

While connected to the Internet, you can upload your package as follows:

```
$ dupload gentoo_0.9.12-1_i386.changes
```

dupload checks that the SHA1/SHA256 file checksums match those listed in the .changes file. If they do not match, it will warn you to rebuild it as described in Section 6.1 so it can be properly uploaded.

If you encounter an upload problem at ftp://ftp.upload.debian.org/pub/UploadQueue/, you can fix this by manually uploading a GPG-signed *.commands file to there with ftp. For example, using hello.commands:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1
Uploader: Foo Bar <Foo.Bar@example.org>
Commands:
  rm hello_1.0-1_i386.deb
  mv hello_1.0-1.dsx hello_1.0-1.dsc
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.10 (GNU/Linux)
-----END PGP SIGNATURE-----
```

1 See Section 1.1.

2 There are publicly accessible archives such as http://mentors.debian.net/ which work almost the same way as the Debian archive and provide an upload area for non-DDs. You can set up an equivalent archive by yourself using the tools listed at http://wiki.debian.org/HowToSetupADebianRepository. So this section is useful for non-DDs, too.

3 The dput package seems to come with more features and to be becoming more popular than the dupload package. It uses the file /etc/dput for its global configuration and the file ~/.dput.cf for per-user configuration. It supports Ubuntu-related services out-of-the-box, too.


5 See ftp://ftp.upload.debian.org/pub/UploadQueue/README. Alternatively, you can use the dcut command from the dput package.
9.2 Including orig.tar.gz for upload

When you first upload the package to the archive, you need to include the original orig.tar.gz source, too. If the Debian revision number of this package is neither 1 nor 0, you must provide the dpkg-buildpackage option -sa.

For the dpkg-buildpackage command:

$ dpkg-buildpackage -sa

For the debuild command:

$ debuild -sa

For the pdebuild command:

$ pdebuild --debbuildopts -sa

On the other hand, the -sd option will force the exclusion of the original orig.tar.gz source.

9.3 Skipped uploads

If you created multiple entries in debian/changelog by skipping uploads, you must create a proper *_.changes file that includes all changes since the last upload. This can be done by specifying the dpkg-buildpackage option -v with the version, e.g., 1.2.

For the dpkg-buildpackage command:

$ dpkg-buildpackage -v1.2

For the debuild command:

$ debuild -v1.2

For the pdebuild command:

$ pdebuild --debbuildopts "-v1.2"
Appendix A

Advanced packaging

Here are some hints and pointers for advanced packaging topics that you are most likely to deal with. You are strongly advised to read all the references suggested here.

You may need to manually edit the packaging template files generated by the `dh_make` command to address topics covered in this chapter. The newer `debmak` command should address these topics better.

A.1 Shared libraries

Before packaging shared libraries, you should read the following primary references in detail:


Here are some oversimplified hints for you to get started:

- Shared libraries are ELF object files containing compiled code.
- Shared libraries are distributed as *.so files. (Neither *.a files nor *.la files)
- Shared libraries are mainly used to share common codes among multiple executables with the `ld` mechanism.
- Shared libraries are sometimes used to provide multiple plugins to an executable with the `dlopen` mechanism.
- Shared libraries export symbols, which represent compiled objects such as variables, functions, and classes; and enable access to them from the linked executables.
- The SONAME of a shared library `libfoo.so.1`: `objdump -p libfoo.so.1 | grep SONAME` ¹
- The SONAME of a shared library usually matches the library file name (but not always).
- The SONAME of shared libraries linked to `/usr/bin/foo`: `objdump -p /usr/bin/foo | grep NEEDED` ²
- `libfoo1`: the library package for the shared library `libfoo.so.1` with the SONAME ABI version 1.³
- The package maintainer scripts of the library package must call `ldconfig` under the specific circumstances to create the necessary symbolic links for the SONAME.⁴

¹ Alternatively: `readelf -d libfoo.so.1 | grep SONAME`
² Alternatively: `readelf -d libfoo.so.1 | grep NEEDED`
• **libfoo1-dbgl** the debugging symbols package that contains the debugging symbols for the shared library package `libfoo1`

• **libfoo-dev** the development package that contains the header files etc. for the shared library `libfoo.so.1`

• Debian packages should not contain `*.la` Libtool archive files in general.

• Debian packages should not use RPATH in general.

• Although it is somewhat outdated and is only a secondary reference, Debian Library Packaging Guide ([http://www.netfort.gr.jp/~dancer/column/libpkg-guide/libpkg-guide.html](http://www.netfort.gr.jp/~dancer/column/libpkg-guide/libpkg-guide.html)) may still be useful.

### A.2 Managing `debian/package.symbols`.

When you package a shared library, you should create a `debian/package.symbols` file to manage the minimal version associated with each symbol for backward-compatible ABI changes under the same SONAME of the library for the same shared library package name. You should read the following primary references in detail:


- `dh_makeshlibs(1)`

- `dpkg-gensymbols(1)`

- `dpkg-shlibdeps(1)`

- `deb-symbols(5)`

Here is a rough example of how to create the `libfoo1` package from the upstream version 1.3 with the proper `debian/libfoo1.symbols` file:

- Prepare the skeleton debianized source tree using the upstream `libfoo-1.3.tar.gz` file.

  - If this is the first packaging of the `libfoo1` package, create the `debian/libfoo1.symbols` file with empty content.

  - If the previous upstream version 1.2 was packaged as the `libfoo1` package with the proper `debian/libfoo1.symbols` in its source package, use it again.

  - If the previous upstream version 1.2 was not packaged with `debian/libfoo1.symbols`, create it as the `symbols` file from all available binary packages of the same shared library package name containing the same SONAME of the library, for example, versions 1.1-1 and 1.2-1.

```
$ dpkg-deb -x libfoo1_1.1-1.deb libfoo1_1.1-1
$ dpkg-deb -x libfoo1_1.2-1.deb libfoo1_1.2-1
$ : > symbols
$ dpkg-gensymbols -v1.1 -Plibfoo1 -Plibfoo1_1.1-1 -Osymbols
$ dpkg-gensymbols -v1.2 -Plibfoo1 -Plibfoo1_1.2-1 -Osymbols
```

- Make trial builds of the source tree with tools such as `debuild` and `pdebuild`. (If this fails due to missing symbols etc., there were some backward-incompatible ABI changes that require you to bump the shared library package name to something like `libfoo1a` and you should start over again.)

---


7See Debian wiki RpathIssue ([http://wiki.debian.org/RpathIssue](http://wiki.debian.org/RpathIssue)).

8Backward-incompatible ABI changes normally require you to update the SONAME of the library and the shared library package name to new ones.


10All previous versions of Debian packages are available at [http://snapshot.debian.org/](http://snapshot.debian.org/). The Debian revision is dropped from the version to make it easier to backport the package: `1.1 << 1.1-1~bpo70+1 << 1.1-1` and `1.2 << 1.2-1~bpo70+1 << 1.2-1`. 

---


$ cd libfoo-1.3
$ debuild
...

dpkg-gensymbols: warning: some new symbols appeared in the symbols file: ...
see diff output below
--- debian/libfoo1.symbols (libfoo1_1.3-1_amd64)
+++ dpkg-gensymbolsFE5gzx 2012-11-11 02:24:53.609667389 +0900
@@ -127,6 +127,7 @@
  foo_get_name@Base 1.1
  foo_get_longname@Base 1.2
+foo_get_longtype@Base 1.3-1
  foo_get_symbol@Base 1.1
  foo_get_rank@Base 1.1
  foo_new@Base 1.1
...

• If you see the diff printed by the *dpkg-gensymbols* as above, extract the proper updated *symbols* file from the generated binary package of the shared library.  

```bash
$ cd ..
$ dpkg-deb -R libfoo1_1.3_amd64.deb libfoo1-tmp
$ sed -e 's/1.3-1/1.3/' libfoo1-tmp/DEBIAN/symbols \ 
>libfoo-1.3/debian/libfoo1.symbols
```

• Build release packages with tools such as *debuild* and *pdebuild*.

```bash
$ cd libfoo-1.3
$ debuild -- clean
$ debuild ...
```

In addition to the above examples, we need to check the ABI compatibility further and bump versions for some symbols manually as needed.  

Although it is only a secondary reference, Debian wiki UsingSymbolsFiles (http://wiki.debian.org/UsingSymbolsFiles) and its linked web pages may be useful.

### A.3 Multiarch

The multiarch feature introduced to Debian wheezy integrates support for cross-architecture installation of binary packages (particularly i386<->amd64, but also other combinations) in dpkg and apt. You should read the following references in detail:

- Ubuntu wiki MultiarchSpec (https://wiki.ubuntu.com/MultiarchSpec) (upstream)
- Debian wiki Multiarch/Implementation (http://wiki.debian.org/Multiarch/Implementation) (Debian situation)

It uses the triplet such as i386-linux-gnu and x86_64-linux-gnu for the install path of shared libraries. The actual triplet path is dynamically set into the `$DEB_HOST_MULTIARCH` variable using the dpkg-architecture(1) command for each binary package build. For example, the path to install multiarch libraries are changed as follows:

```bash
The Debian revision is dropped from the version to make it easier to backport the package: 1.3 << 1.3-1-bpo70+1 << 1.3-1
```
Here are some typical multiarch package split scenario examples for the following:

- a library source `libfoo-1.tar.gz`
- a tool source `bar-1.tar.gz` written in a compiled language
- a tool source `baz-1.tar.gz` written in an interpreted language

<table>
<thead>
<tr>
<th>Package</th>
<th>Architecture:</th>
<th>Multi-Arch:</th>
<th>Package content</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>libfoo</code></td>
<td>any</td>
<td>same</td>
<td>the shared library, co-installable</td>
</tr>
<tr>
<td><code>libfoo-db</code></td>
<td>any</td>
<td>same</td>
<td>the shared library debug symbols, co-installable</td>
</tr>
<tr>
<td><code>libfoo-dev</code></td>
<td>any</td>
<td>same</td>
<td>the shared library header files etc., co-installable</td>
</tr>
<tr>
<td><code>libfoo-tools</code></td>
<td>any</td>
<td>foreign</td>
<td>the run-time support programs, not co-installable</td>
</tr>
<tr>
<td><code>libfoo-doc</code></td>
<td>all</td>
<td>foreign</td>
<td>the shared library documentation files</td>
</tr>
<tr>
<td><code>bar</code></td>
<td>any</td>
<td>foreign</td>
<td>the compiled program files, not co-installable</td>
</tr>
<tr>
<td><code>bar-doc</code></td>
<td>all</td>
<td>foreign</td>
<td>the documentation files for the program</td>
</tr>
<tr>
<td><code>baz</code></td>
<td>all</td>
<td>foreign</td>
<td>the interpreted program files</td>
</tr>
</tbody>
</table>

Please note that the development package should contain a symlink for the associated shared library **without a version number**. E.g.: `/usr/lib/x86_64-linux-gnu/libfoo.so` -> `libfoo.so.1`

### A.4 Building a shared library package

You can build a Debian library package enabling multiarch support using `dh(1)` as follows:

- Update `debian/control`.
  - Add `Build-Depends: debhelper (>=10)` for the source package section.
  - Add `Pre-Depends: ${misc:Pre-Depends}` for each shared library binary package.
  - Add `Multi-Arch: stanza` for each binary package section.
- Set `debian/compat` to "10".
- Adjust the path from the normal `/usr/lib/` to the multiarch `/usr/lib/$(DEB_HOST_MULTIARCH)/` for all packaging scripts.
  - Call `DEB_HOST_MULTIARCH` `=$(shell dpkg-architecture -qDEB_HOST_MULTIARCH)` in `debian/rules` to set the `DEB_HOST_MULTIARCH` variable first.
  - Replace `/usr/lib/` with `/usr/lib/$(DEB_HOST_MULTIARCH)/` in `debian/rules`.
  - If `/configure` is used in part of the `override_dh_auto_configure` target in `debian/rules`, make sure to replace it with `dh_auto_configure --`.  
  - Replace all occurrences of `/usr/lib/` with `/usr/lib/*` in `debian/foo.install` files.
  - Generate files like `debian/foo.links` from `debian/foo.links.in` dynamically by adding a script to the `override_dh_auto_configure` target in `debian/rules`.

14Alternatively, you can add `--libdir=${prefix}/lib/$(DEB_HOST_MULTIARCH)` and `--libexecdir=${prefix}/lib/$(DEB_HOST_MULTIARCH)` arguments to `/configure`. Please note that `--libexecdir` specifies the default path to install executable programs run by other programs rather than by users. Its Autotools default is `/usr/libexec/` but its Debian default is `/usr/lib/`.
override dh_auto_configure:
    dh_auto_configure
    sed 's/@DEB_HOST_MULTIARCH@/$(DEB_HOST_MULTIARCH)/g' \
        debian/foo.links.in > debian/foo.links

Please make sure to verify that the shared library package contains only the expected files, and that your -dev package still works. All files installed simultaneously as the multiarch package to the same file path should have exactly the same file content. You must be careful of differences generated by the data byte order and by the compression algorithm.

A.5 Native Debian package

If a package is maintained only for Debian or possibly only for local use, its source may contain all the debian/* files in it. There are 2 ways to package it.

You can make the upstream tarball by excluding the debian/* files and package it as a non-native Debian package as in Section 2.1. This is the normal way, which some people encourage using.

The alternative is the workflow of the native Debian package.

• Create a native Debian source package in the 3.0 (native) format using a single compressed tar file in which all files are included.
  – package_version.tar.gz
  – package_version.dsc

• Build Debian binary packages from the native Debian source package.
  – package_version_arch.deb

For example, if you have source files in ~/mypackage-1.0 without the debian/* files, you can create a native Debian package by issuing the dh_make command as follows:

$ cd ~/mypackage-1.0
$ dh_make --native

Then the debian directory and its contents are created just like in Section 2.8. This does not create a tarball, since this is a native Debian package. But that is the only difference. The rest of the packaging activities are practically the same.

After execution of the dpkg-buildpackage command, you will see the following files in the parent directory:

• mypackage_1.0.tar.gz
  This is the source code tarball created from the mypackage-1.0 directory by the dpkg-source command. (Its suffix is not orig.tar.gz.)

• mypackage_1.0.dsc
  This is a summary of the contents of the source code, as in the non-native Debian package. (There is no Debian revision.)

• mypackage_1.0_i386.deb
  This is your completed binary package, as in the non-native Debian package. (There is no Debian revision.)

• mypackage_1.0_i386.changes
  This file describes all the changes made in the current package version as in the non-native Debian package. (There is no Debian revision.)