

# Debian New Maintainers' Guide

Josip Rodin <joy-mg@debian.org>

version 1.2.13, 5 June 2008.

## **Copyright Notice**

Copyright © 1998-2002 Josip Rodin.

Copyright © 2005-2007 Osamu Aoki.

This document may be used under the terms the GNU General Public License version 2 or higher.

This document was made using with these two documents as examples:

Making a Debian Package (AKA the Debmake Manual), copyright © 1997 Jaldhar Vyas.

The New-Maintainer's Debian Packaging Howto, copyright © 1997 Will Lowe.

---

# Contents

<b>1</b>	<b>Getting started The Right Way</b>	<b>1</b>
1.1	Programs you need for development . . . . .	1
1.2	Official Debian Developer . . . . .	3
1.3	Other information . . . . .	4
<b>2</b>	<b>First steps</b>	<b>5</b>
2.1	Choose your program . . . . .	5
2.2	Get the program, and try it out . . . . .	6
2.3	Package name and version . . . . .	7
2.4	Initial “debianization” . . . . .	7
<b>3</b>	<b>Modifying the source</b>	<b>9</b>
3.1	Installation in a subdirectory . . . . .	9
3.2	Differing libraries . . . . .	12
<b>4</b>	<b>Required stuff under debian/</b>	<b>13</b>
4.1	‘control’ file . . . . .	13
4.2	‘copyright’ file . . . . .	17
4.3	‘changelog’ file . . . . .	20
4.4	‘rules’ file . . . . .	21
<b>5</b>	<b>Other files under debian/</b>	<b>23</b>
5.1	README.Debian . . . . .	23
5.2	conffiles . . . . .	24
5.3	Cron files . . . . .	24

---

5.4	<code>dirs</code>	25
5.5	<code>docs</code>	25
5.6	<code>emacsen-*.ex</code>	25
5.7	<code>files</code>	26
5.8	<code>init.d</code> and <code>package.default</code>	26
5.9	<code>manpage.1.ex</code> , <code>manpage.sgml.ex</code> , <code>manpage.xml.ex</code>	26
5.10	<code>menu.ex</code>	28
5.11	<code>package.doc-base.EX</code>	29
5.12	<code>postinst.ex</code> , <code>preinst.ex</code> , <code>postrm.ex</code> , <code>prerm.ex</code>	29
5.13	<code>watch.ex</code>	29
5.14	<code>source/format</code>	30
<b>6</b>	<b>Building the package</b>	<b>31</b>
6.1	Complete rebuild	31
6.2	Quick rebuild	32
6.3	The <code>debuild</code> command	33
6.4	The <code>dpatch</code> and <code>quilt</code> systems	33
6.5	Including <code>orig.tar.gz</code> for upload	34
<b>7</b>	<b>Checking the package for errors</b>	<b>35</b>
7.1	The <code>lintian</code> package	35
7.2	The <code>mc</code> command	35
7.3	The <code>debdiff</code> command	36
7.4	The <code>interdiff</code> command	36
7.5	The <code>debi</code> command	36
7.6	The <code>pbuilder</code> package	36
<b>8</b>	<b>Uploading the package</b>	<b>39</b>
8.1	Uploading to the Debian archive	39
8.2	Uploading to a private archive	40

---

<b>9</b>	<b>Updating the package</b>	<b>43</b>
9.1	New Debian revision . . . . .	43
9.2	New upstream release (basic) . . . . .	43
9.3	New upstream release (realistic) . . . . .	44
9.4	The <code>orig.tar.gz</code> file . . . . .	46
9.5	The <code>cvs-buildpackage</code> command and similes . . . . .	46
9.6	Verifying package upgrades . . . . .	46
<b>10</b>	<b>Where to ask for help</b>	<b>49</b>
<b>A</b>	<b>Examples</b>	<b>51</b>
A.1	Simple packaging example . . . . .	51
A.2	Packaging example with the <code>dpatch</code> and the <code>pbuilder</code> . . . . .	51



# Chapter 1

## Getting started The Right Way

This document tries to describe building of a Debian package to the common Debian user, and prospectus developer. It uses pretty common language, and it's well covered with working examples. There is an old Roman 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!).

One of the things that makes Debian such a top-notch Linux 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 rookie, you wouldn't be reading this doc now. :-) You do need to know a little about Unix programming but you certainly don't need to be a wizard.

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

This document will explain every little (at first maybe irrelevant) step, and help you create that first package, and to gain some experience in building next releases of that and maybe other packages later on.

Newer versions of this document should always be available online at <http://www.debian.org/doc/maint-guide/> and in the 'maint-guide' package.

### 1.1 Programs you need 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.

This revision of this document has been updated for the packages in Debian 5.0 ('lenny').

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

with `'dpkg -s <package>'`.

The most important package to install on your system is the `build-essential` package. Once you try to install it, 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 install or may be required by your package:

- `file` - this handy program can determine what type a file is. (see `file(1)`)
- `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)`)
- `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)`)
- `autoconf`, `automake` and `autotools-dev` - many newer programs use configure scripts and Makefiles preprocessed with help of programs like these. (see ‘info autoconf’, ‘info automake’). The `autotools-dev` keeps up-to-date versions of certain auto files and has documentation about the best way to use those files.
- `dh-make` and `debhelper` - `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 creation of packages, but are **highly** recommended for new maintainers. It makes the whole process very much easier to start, and control afterwards. (see `dh_make(1)`, `debhelper(1)`, `/usr/share/doc/debhelper/README`)
- `devscripts` - this package contains some nice and useful scripts that can be helpful to the maintainers, but they are also not necessary for building packages. (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)`)
- `gnupg` - a tool that enables you to digitally *sign* packages. This is especially important if you want to distribute it to other people, and you will certainly be doing that when your work gets included in the Debian distribution. (see `gpg(1)`)
- `gfortran` - the GNU Fortran 95 compiler, necessary if your program is written in Fortran. (see `gfortran(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)`)
- `xutils` - some programs, usually those made for X11, also use these programs to generate Makefiles from sets of macro functions. (see `imake(1)`, `xmkmf(1)`)

- `lintian` - this is the Debian package checker that can let you know of any common mistakes after you build the package, and explain the errors found. (see `lintian(1)`, `/usr/share/doc/lintian/lintian.html/index.html`)
- `pbuilder` - this package contains programs which is used for creating and maintaining chroot environment. Building Debian package in this chroot environment verifies the proper build dependency and avoid FTBFS bugs. (see `pbuilder(8)` and `pdebuild(1)`)

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

- `debian-policy` - the Policy includes explanations of the structure and contents of the Debian archive, several OS design issues, the Filesystem Hierarchy Standard (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 `/usr/share/doc/debian-policy/policy.html/index.html`)
- `developers-reference` - for all matters not specifically about the technical details of packaging, like the structure of the archive, how to rename, orphan, pick up packages, how to do NMUs, how to manage bugs, best packaging practices, when and where to upload etc. (see `/usr/share/doc/developers-reference/index.en.html`)

The short descriptions that are given above only serve to introduce you to what each package does. Before continuing please thoroughly read the documentation of each program, at least the standard usage. It may seem like heavy going now, but later on you'll be *very* glad you read it.

## 1.2 Official Debian Developer

After you build your package (or while doing that), you may want to become an official Debian Developer to get your new package into the next distribution (if the program is useful, why not?).

You can not become an official Debian Developer over night because it takes more than technical skill. Please do not be discouraged by this. You can still upload your package, if useful to others, now as a maintainer through a sponsor while applying yourself to the Debian New Maintainer process (<http://nm.debian.org/>). Here, the sponsor is an official Debian Developer who helps maintainer to upload packages to the Debian archive. More details of this procedure are explained in the debian-mentors FAQ ([http://people.debian.org/~mpalmer/debian-mentors\\_FAQ.html](http://people.debian.org/~mpalmer/debian-mentors_FAQ.html)).

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 become an official Debian Developer too.

### 1.3 Other information

There are two types of packages you can make, source and binary. A source package contains code which you can compile into a program. A binary package contains just the finished program. Don't mix terms like source of the program and the source package of the program! Please read the other manuals if you need more details on terminology.

In Debian, the term 'maintainer' is used for the person who makes packages, 'upstream author' for the person that made the program, and 'upstream maintainer' for the person who currently maintains that program, outside of Debian. Usually author and the upstream maintainer are the same person - and sometimes even the maintainer is the same person. If you made a program, and want it to get in Debian, feel free to submit your application to become a maintainer.

## Chapter 2

# First steps

### 2.1 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 `aptitude`. If you use the 'stable' distribution, maybe it's best that you go to the package search page (<http://www.debian.org/distrib/packages>).

If the package already exists, well, install it! :-) If it happens to be orphaned – if its maintainer is set to “Debian QA Group”, you may be able to pick it up.

Then you should consult the Debian web site at Work-Needing and Prospective Packages (<http://www.debian.org/devel/wnpp/>) and its linked pages to check the latest adoption/orphan status of the package.

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 set up for you. Keep reading, though, a lot of the advice below will still be applicable for your case.

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

- check if no one else is working on the package already at the list of packages being worked on ([http://www.de.debian.org/devel/wnpp/being\\_packaged](http://www.de.debian.org/devel/wnpp/being_packaged)). If someone's already on it, contact them if you feel you need to. If not - find another interesting program that nobody maintains.
- program **must** have a license.
  - For the `main` section, it must be compliant to all the Debian Free Software Guidelines ([http://www.debian.org/social\\_contract#guidelines](http://www.debian.org/social_contract#guidelines)) (DFSG) and **must not** require a package outside of `main` for compilation or execution as required by the Debian Policy. This is desired case.

- For the `contrib` section, it **must** be compliant to all the DSFG but it may require a package outside of `main` for compilation or execution.
- For the `non-free` section, it **may not** be compliant to some of the DSFG but it **must** be distributable.

If you are unsure about where it should go, post the license text on `<debian-legal@lists.debian.org>` and ask for advice.

- program certainly should **not** run `setuid root`, or even better - it shouldn't need to be `setuid` or `setgid` to anything.
- program should not be a daemon, or something that goes in `*/sbin` directories, or open a port as root.
- program should result in binary executable form, libraries are harder to handle.
- it should be well documented, or and the code needs to be understandable (i.e. not obfuscated).
- you should contact program's author(s) to check if they agree with packaging it. It is important to be able to consult with author(s) about the program in case of any program specific problems, so don't try to package unmaintained pieces of software.
- and last but not least, you must know that it works, and have tried it for some time.

Of course, these things are just safety measures, and intended to save you from raging users if you do something wrong in some `setuid` daemon... When you gain some more experience in packaging, you'll be able to do such packages, but even the experienced developers consult the `debian-mentors` mailing list when they are in doubt. And people there will gladly help.

For more help about these, check in `Developer's Reference`.

## 2.2 Get the program, and try it out

So the first thing to do is to find and download the original package. I presume that 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 extension `.tar.gz`, and usually contain the subdirectory called `program-version` and all the sources in it. If your program's source comes as some other sort of archive (for instance, the filename ends in `".Z"` or `".zip"`), unpack it with appropriate tools, or ask on the `debian-mentors` mailing list if you're not sure how to unpack it correctly (hint: issue `'file archive.extension'`).

As an example, I'll use a program called `'gentoo'`, an X GTK+ file manager. Note that the program is already packaged, and has changed substantially from the version while this text was first written.

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 uncompress it (with `'tar xzf gentoo-0.9.12.tar.gz'`). Make sure there are no errors, even some “irrelevant” ones, because there will most probably be problems unpacking on other people’s systems, whose unpacking tools may or may not ignore those anomalies.

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 correctly compile and install the program (most probably they’ll assume you want to install to `/usr/local/bin` directory; you won’t be doing that, but more on that later in ‘Installation in a subdirectory’ on page 9).

The process varies from program to program, but a lot of modern programs come with a `'configure'` script that configures the source under your system and makes sure that your system is in condition to compile it. After configuring with `'./configure'`, programs are usually compiled with `'make'`. Some of them support `'make check'`, to run included self-checks. Installation in 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.

## 2.3 Package name and version

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

For the package to be built correctly, you must make the program’s original name lowercase (if it isn’t already), and you should move the source directory to `<packagename>-<version>`.

If the program name consists of more than one word, contract them to one word, or make an abbreviation. For example, program “John’s little editor for X” package would be named `johnledx`, or `jle4x`, or whatever you decide, as long as it’s under some reasonable limit, e.g. 20 characters.

Also check for the exact version of the program (to be included in the package version). If that piece of software is not numbered with versions like `X.Y.Z`, but with some kind of date, feel free to use that date as the version number, as long as newer version numbers will look larger. While it is best to use the same version number as what upstream uses, if it is in the format of `09Oct23` you may need to convert it to `YYYYMMDD` format, which would be `20091023`.

Some programs won’t be numbered at all, in which case you should contact the upstream maintainer to see if they’ve got some other revision-tracking method.

## 2.4 Initial “debianization”

Make sure you’re in the program source directory, and issue this:

```
dh_make -e your.maintainer@address -f ../gentoo-0.9.12.tar.gz
```

Of course, replace the string “your.maintainer@address” with your e-mail address for inclusion in the changelog entry and other files, and the filename with the name of your original source archive. See `dh_make(1)` for details.

Some information will come up. It will ask you what sort of package you want to create. Gentoo is a single binary package - it creates only one binary, and thus 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>.

After this execution of `dh_make`, a copy of the upstream tarball is created 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 `diff.gz`. Please note 2 key features in this file name:

- Package name and version are separated by the “\_” .
- There is the “orig.” before the “tar.gz” .

Once again, as a new maintainer you are discouraged from creating complicated packages, e.g.,

- multiple binary packages,
- library packages,
- the source file format being neither in `tar.gz` . nor `tar.bz2`, or
- the source tarball containing undistributable contents.

It’s not too hard, but it does require a bit more knowledge, so we won’t describe all of it here.

Please note that you should run `dh_make` **only once**, and that it won’t behave correctly if you run it again in the same, already “debianized”, directory. That also means that you will use a different method to release a new revision or a new version of your package in the future. Read more about that later in ‘Updating the package’ on page [43](#)

---

## Chapter 3

# Modifying the source

Normally, programs install themselves in the `/usr/local` subdirectories. But, Debian packages must not use that directory, since it is reserved for system administrator's (or user's) private use. This means that you have to take a look at your program's build system, usually starting with the Makefile. This is the script `make(1)` will use to automate building this program. For more details on Makefiles, look in "rules' file' on page 21.

Note that if your program uses GNU `automake(1)` and/or `autoconf(1)`, meaning the source includes `Makefile.am` and/or `Makefile.in` files, respectively, you will need to modify those files. This is because each `automake` invocation will rewrite `Makefile.in`'s with information generated from `Makefile.am`'s, and each `./configure` invocation will do the same with `Makefile`'s, with data from `Makefile.in`'s. Editing `Makefile.am` files requires some knowledge of `automake`, which you can read about in the `automake info` entry, whereas editing `Makefile.in` files is pretty much the same as editing `Makefile` files, just pay attention to the variables, i.e. any strings surrounded with '@'s, for example `@CFLAGS@` or `@LN_S@`, which are replaced with actual stuff on each `./configure` invocation. Please make sure to read `/usr/share/doc/autotools-dev/README.Debian.gz` before proceeding.

Also note that there isn't space here to go into *all* the details of fixing upstream sources, but here are a few problems people often run across.

### 3.1 Installation in a subdirectory

Most of the programs have some way of installing themselves in the existing directory structure of your system, so that their binaries get included in your `$PATH`, and that you find their documentation and manual pages in common places. However, if you do that, the program will be installed among everything else already on your system. This would make it hard for the packaging tools to figure out which files belong to your package and which don't.

Therefore you need to do something else: install the program into a temporary subdirectory from which the maintainer tools will build a working `.deb` package. Everything that is contained in this directory will be installed on a user's system when they install your package, the only difference is that `dpkg` will be installing the files in the root directory.

This temporary directory is usually created under your `debian/` directory in the unpacked source tree. It is usually called `debian/packageName`.

Bear in mind that even though you need to make the program install in `debian/packageName`, it still needs to behave correctly when placed in the root directory, i.e. when installed from the `.deb` package. So you mustn't allow the build system to hardcode strings like `/home/me/deb/gentoo-0.9.12/usr/share/gentoo` into the package files.

With programs that use GNU autoconf, this will be quite easy. Most such programs have makefiles that are by default set up to allow installation into a random subdirectory while keeping in mind that `/usr` (for example) is the canonical prefix. When it detects your program uses autoconf, `dh_make` will set up commands for doing all this automatically, so you might as well skip reading this section. But with other programs, you will most probably have to examine and edit the Makefiles.

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

```
# Where to put binary 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`. Change those paths to:

```
# Where to put binary on 'make install'?
BIN      = $(DESTDIR)/usr/bin

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

But why in that directory, and not some other? Because Debian packages never install files beneath `/usr/local` – that tree is reserved for the system administrator's use. Such files on Debian systems go under `/usr` instead.

The more exact locations for binaries, icons, documentation etc are specified in the Filesystem Hierarchy Standard (see `/usr/share/doc/debian-policy/fhs/`). I recommend you browse it and read the sections that might concern your package.

So, we should install the binary in `/usr/bin` instead of `/usr/local/bin`, the manual page in `/usr/share/man/man1` instead of `/usr/local/man/man1` etc. Notice how there's no manual page mentioned in gentoo's makefile, but since the 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 -e 'usr/local/lib' --include='*.[c|h]' .
```

Grep will run recursively through the source tree and tell you the name of the file and the line in it, when it finds an occurrence.

Edit those files and in those lines replace `/usr/local/*` with `usr/*` – and that’s about it. Be careful that you don’t mess up the rest of the code! :-)

After that you should find the `install` target (search for line that starts with `install:`, that will usually work) and rename all references to directories other than ones defined at the top of the Makefile. Previously, gentoo’s `install` target said:

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

After our change it says:

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

You’ve surely noticed that there’s now a `install -d` command before the other commands in the rule. The original makefile didn’t have it because usually the `/usr/local/bin` and other directories already exist on the system where one runs `make install`. However, since we will install into our own empty (or even nonexistent) directory, 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:

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

A careful reader will notice that I changed `gentoo` to `gentoo-target` in the `install:` line. That is called an unrelated bug fix :-)

Whenever you make changes that are not specifically related to Debian package, be sure to send them to the upstream maintainer so they can be included in the next program revision and be useful to everyone else. Also remember to make your fixes not specific to Debian or Linux (or even Unix!) prior to sending them – make them portable. This will make your fixes much easier to apply.

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

## 3.2 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 which doesn't exist on Debian systems. In that case, we need to change it to a library which does exist in Debian, and serves the same purpose.

So, if there is a line in your program's Makefile (or Makefile.in) that says something like this (and your program doesn't compile):

```
LIBS = -lcurses -lsomething -lsomethingelse
```

Change it to this, and it will most probably work:

```
LIBS = -lncurses -lsomething -lsomethingelse
```

(The author realizes that this is not the best example considering our libncurses package now ships with a libcurses.so symlink, but he couldn't think of a better one. Suggestions very welcome :-)

## Chapter 4

# Required stuff under debian/

There is a new subdirectory under the program's source directory, it's 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.

### 4.1 'control' file

This file contains various values which `dpkg`, `dselect` and other package management tools will use to manage the package.

Here is the control file `dh_make` created for us:

```
1 Source: gentoo
2 Section: unknown
3 Priority: extra
4 Maintainer: Josip Rodin <joy-mg@debian.org>
5 Build-Depends: debhelper (>= 7)
6 Standards-Version: 3.8.3
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 1-6 are the control information for the source 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, Debian is divided in sections: main (the free software), non-free (the not really free software) and contrib (free software that depends on non-free software). Under those, there are logical subsections that describe in short what packages are in. So we have 'admin' for administrator-only programs, 'base' for the basic tools, 'devel' for programmer tools, 'doc' for documentation, 'libs' for libraries, 'mail' for e-mail readers and daemons, 'net' for network apps and daemons, 'x11' for X11 programs that don't fit anywhere else, and many more.

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. See the Policy manual for guidance on what to set this field to. The "extra" priority will usually work for new packages.

Section and priority are used by frontends like `dselect` 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 an email, because after you upload it, the bug tracking system will use it to deliver bug emails to you. Avoid using commas, ampersands and parenthesis.

The 5th line includes the list of packages required to build your package. Some packages like `gcc` and `make` are implied, see the `build-essential` package for details. If some non-standard compiler or other tool is needed to build your package, you should add it to the 'Build-Depends' line. Multiple entries are separated with commas; read on for the explanation of binary dependencies to find out more about the syntax of this field.

You can also have `Build-Depends-Indep`, `Build-Conflicts` and other fields here. This data will be used by the Debian automatic package building software in order to create binary packages for other computer platforms. See the Policy manual for more information about the build-dependencies and the Developers' Reference for more information about these other platforms (architectures) and how to port software to them.

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

```
dpkg-depcheck -d ./configure
```

To manually find exact build dependency for `/usr/bin/foo`, you 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 you just take `-dev` version of every package as 'Build-deps' entry. If you use `ldd` for this purpose, it will report indirect lib dependencies as well, resulting in the problem of excessive build deps.

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`.

Line 6 is the version of the Debian Policy standards this package follows, the versions of the Policy manual you read while making your package.

On line 7 you can put the URL of the homepage for the upstream package.

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 CPU architecture the binary package can be compiled for. We'll leave this as "any" because `dpkg-gencontrol(1)` will fill in the appropriate value for any machine this 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 "rules' file' on page 21 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:`, `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.)

This is what the dependencies mean:

- **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:**

Frontends such as `dselect` or `aptitude` will prompt you to install the recommended packages along with your package; `dselect` will even insist. `dpkg` and `apt-get` will ignore this field, though. Use this for packages that are not strictly necessary but are typically used with your program.

- **Suggests:**

When a user installs your program, all frontends will likely prompt them to install the suggested packages. `dpkg` and `apt-get` won't care. Use this for packages which will work nicely with your program but are not at all necessary.

- **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 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.

- Provides:

For some types of packages where there are multiple alternatives virtual names have been defined. You can get the full list in the /usr/share/doc/debian-policy/virtual-package-names-list.txt.gz 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. These versions are listed in parentheses after each individual package name, and they should contain a relation from the list below followed by the version number. The relations allowed are: <<, <=, =, >= and >> for strictly earlier, earlier or equal, exactly equal, later or equal and strictly later, respectively. For example,

```
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}. After your package has been built and installed into the temporary directory, dh\_shlibdeps(1) will scan it for binaries and libraries, determine their shared library dependencies and detect which packages they are in, such as libc6 or xlib6g. It'll pass on the list to dh\_gencontrol(1) which will fill it in the right place, and you won't have to worry about this yourself.

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

Line 12 is the short description. Most people screens are 80 columns wide so this shouldn't be longer than about 60 characters. I'll change it to "fully GUI configurable X file manager using GTK+".

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.

Finally, here is the updated control file:

```
1 Source: gentoo
2 Section: x11
3 Priority: optional
4 Maintainer: Josip Rodin <joy-mg@debian.org>
5 Build-Depends: debhelper (>> 3.0.0), xlibs-dev, libgtk1.2-dev, libglib1.
6 Standards-Version: 3.8.3
7 Homepage: http://www.obsession.se/gentoo/
8
9 Package: gentoo
10 Architecture: any
11 Depends: ${shlibs:Depends}, ${misc:Depends}
12 Suggests: file
13 Description: fully GUI configurable X file manager using GTK+
14 gentoo is a file manager for Linux written from scratch in pure C. It
15 uses the GTK+ toolkit for all of its interface needs. gentoo provides
16 100% GUI configurability; no need to edit config files by hand and re-
17 start the program. gentoo supports identifying the type of various
18 files (using extension, regular expressions, or the 'file' command),
19 and can display files of different types with different colors and icon
20 .
21 gentoo borrows some of its look and feel from the classic Amiga file
22 manager "Directory OPUS" (written by Jonathan Potter).
```

(I've added the line numbers.)

## 4.2 'copyright' file

This file contains the information about package upstream resources, copyright and license information. Its format is not dictated by the Policy, but the content is (section 12.6 "Copyright information").

dh\_make can give you a template copyright file, use the `-copyright` option to select the right template. As gentoo is licensed under the GPLv2 license, using the `-copyright gpl2` option will give the following:

```
1 This work was packaged for Debian by:
2
3     Josip Rodin <joy-mg@debian.org> on Wed, 11 Nov 1998 21:02:14 +0100.
4
5 It was downloaded from:
6
7     <url://example.com>
8
9 Upstream Author(s):
10
11     <put author's name and email here>
12
13 Copyright:
14
15     <Copyright (C) YYYY Firstname Lastname>
16     <likewise for another author>
17
18 License:
19
20 ### SELECT: ###
21     This package is free software; you can redistribute it and/or modify
22     it under the terms of the GNU General Public License as published by
23     the Free Software Foundation; either version 2 of the License, or
24     (at your option) any later version.
25 ### OR ###
26     This package is free software; you can redistribute it and/or modify
27     it under the terms of the GNU General Public License version 2 as
28     published by the Free Software Foundation.
29 #####
30
31     This package is distributed in the hope that it will be useful,
32     but WITHOUT ANY WARRANTY; without even the implied warranty of
33     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
34     GNU General Public License for more details.
35
36     You should have received a copy of the GNU General Public License
37     along with this package; if not, write to the Free Software
38     Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-130
39
40 Debian systems, the complete text of the GNU General
41 License version 2 can be found in '/usr/share/common-licenses/GPL-2'.
42
43 Debian packaging is:
44
45     Copyright (C) 1998 Josip Rodin <joy-mg@debian.org>
46
```

```
47 You can redistribute it and/or modify
48 it under the terms of the GNU General Public License as published by
49 Free Software Foundation; either version 2 of the License, or
50 (at your option) any later version.
51
52 # Please also look if there are files or directories which have a
53 # different copyright/license attached and list them here.
```

(I've added the line numbers.)

The important things to add to this file are the place you got the package from and the actual copyright notice and license. You must include the complete license, unless it's one of the common free software licenses such as GNU GPL or LGPL, BSD or the Artistic license, when you can just refer to the appropriate file in /usr/share/common-licenses/ directory that exists on every Debian system.

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

```
1 This work was packaged for Debian by:
2
3     Josip Rodin <joy-mg@debian.org> on 2   Wed, 11 Nov 1998 21:02:14 +0100
4
5 It was downloaded from:
6     ftp://ftp.obsession.se/gentoo/
7
8 Upstream author:
9
10    Emil Brink <emil@obsession.se>
11
12 Copyright:
13    Copyright (C) 1998-99 by Emil Brink, Obsession Development.
14
15 License:
16    You are free to distribute this software under the terms of
17    the GNU General Public License either version 2 of the License,
18    or (at your option) any later version.
19
20    This package is distributed in the hope that it will be useful,
21    but WITHOUT ANY WARRANTY; without even the implied warranty of
22    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
23    GNU General Public License for more details.
24
25    You should have received a copy of the GNU General Public License
26    along with this package; if not, write to the Free Software
27    Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1300
28
```

```
29 On Debian systems, the complete text of the GNU General
30 Public License version 2 can be found in '/usr/share/common-licenses/GPL
31
32 The Debian packaging is:
33
34     Copyright (C) 1998 Josip Rodin <joy-mg@debian.org>
35
36 You can redistribute it and/or modify
37 it under the terms of the GNU General Public License as published by
38 Free Software Foundation; either version 2 of the License, or
39 (at your option) any later version.
```

(I've added the line numbers.)

Please follow the HOWTO from the debian-devel-announce: <http://lists.debian.org/debian-devel-announce/2006/03/msg00023.html>.

### 4.3 'changelog' file

This is a required file, which has a special format described in the Policy section 4.4 "debian/changelog". 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 how it looks like:

```
1 gentoo (0.9.12-1) unstable; urgency=low
2
3   * Initial Release.
4
5  -- Josip Rodin <joy-mg@debian.org>   Wed, 11 Nov 1998 21:02:14 +0100
6
```

(I've added the line numbers.)

Line 1 is the package name, version, distribution, and urgency. The name must match the source package name, distribution should be either 'unstable' (or even 'experimental'), and urgency shouldn't be changed to anything higher than 'low'. :-)

Lines 3-5 are a log entry, where you document changes made in this package revision (not the upstream changes - there is special file for that purpose, created by the upstream authors,

which you will later install as `/usr/share/doc/gentoo/changelog.gz`). New lines must be inserted just before the uppermost line that begins with asterisk (\*). You can do it with `dch(1)`, or manually with a text editor.

You will end up with something like this:

```
1 gentoo (0.9.12-1) unstable; urgency=low
2
3  * Initial Release.
4  * This is my first Debian package.
5  * Adjusted the Makefile to fix $DESTDIR problems.
6
7  -- Josip Rodin <joy-mg@debian.org> Wed, 11 Nov 1998 21:02:14 +0100
8
```

(I've added the line numbers.)

You can read more about updating the changelog file later in 'Updating the package' on page 43.

## 4.4 'rules' file

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

Every 'rules' file, as any other Makefile, consists of several rules specifying how to handle the source. Each rule consists of targets, filenames or names of actions that should be carried out (e.g. 'build:' or 'install:'). Rules that you want to execute are invoked as command line arguments (for example, `./debian/rules build` or `make -f rules install`). After the target name, you can name the dependency, program or file that the rule depends on. After that, there can be any number of commands, indented with `<tab>`. A new rule begins with the target declaration in the first column. Empty lines and lines beginning with '#' (hash) are treated as comments and ignored.

You are probably confused now, but it will all be clear upon examination of the 'rules' file that `dh_make` gives us as a default. You should also read the 'make' entry in `info` for more information.

The important part to know about the rules file created by `dh_make`, is that it is just a suggestion. It will work for simple packages but for more complicated ones, don't be afraid to add and subtract from it to fit your needs. Only thing that you must not change are the names of the rules, because all the tools use these names, as mandated by the Policy.

Here's (approximately) how the default `debian/rules` file that `dh_make` generated for us looks like:

```
1  #!/usr/bin/make -f
2  # -*- makefile -*-
3  # Sample debian/rules that uses debhelper.
4  # This file was originally written by Joey Hess and Craig Small.
5  # As a special exception, when this file is copied by dh-make into a
6  # dh-make output file, you may use that output file without restriction.
7  # This special exception was added by Craig Small in version 0.37 of dh-
8
9  # Uncomment this to turn on verbose mode.
10 #export DH_VERBOSE=1
11
12 %:
13     dh $@
```

(I've added the line numbers. In the actual `debian/rules` file, the leading white spaces are TAB codes.)

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 10 can be uncommented which will mean `DH_VERBOSE` variable is set to 1. This will mean the debhelper tools will output more information which can be helpful in debugging what is going wrong.

Lines 12 and 13 are where all the work is done. The percent sign means any targets which then call a single program, `dh` with the target name. For example, when `dpkg-buildpackage` wants to do the pre-build cleaning, it runs `debian/rules clean` which in turn calls “`dh clean`”.

For more complete information on what do all these `dh_*` scripts do, and what their other options are, please read their respective manual pages. There are some other (possibly very useful) `dh_*` scripts which were not mentioned here. If you need them, read the debhelper documentation.

The major change between older rules files that used debhelper and the current ones, is that the old ones had very large rules file which called each `dh_*` script, while the new ones have a single line calling `dh`.

Previously, if you wanted to change the behaviour of a `dh_*` script you found the relevant line in the rules file and adjusted it. Now you use overrides lines in the rules file, only for the command you want to change.

Override lines are targets that basically say “run me instead”. If you have a target for `override_dh_foo` then instead of `dh_foo` being run, your commands for that target are run instead.

Gentoo has an unusual upstream changelog file called `FIXES`. Now, `dh_installchangelogs` will not install that file by default, so we need to change what is run at this point and the following lines need to be appended to the rules file:

```
override_dh_installchangelogs:
    dh_installchangelogs FIXES
```

## Chapter 5

# Other files under debian/

To control most of what debhelper is doing when it builds the package, you put files under the debian directory. This chapter will describe what each of the files does and its format.

dh\_make will create some example files in this directory, you will see that there are several other files in the debian/ subdirectory, most of them with the 'ex' suffix or prefix, meaning that they are examples. Take a look at all of them. If you wish or need to use any of those features:

- take a look at the related documentation (hint: the Policy Manual),
- if necessary modify the files to suit your needs,
- rename them to remove the '.ex' suffix if they have one,
- rename them to remove the 'ex.' prefix if they have one,
- modify the 'rules' file if necessary.

Some of those files, the commonly used ones, are explained in the following sections. File-names can either start with the package name, for example gentoo.dirs or not. If there is no package name then the first binary package uses these files, if there is no same filename prepended with package.

### 5.1 README.Debian

Any extra details or discrepancies between the original package and your debianized 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
```

Since we don't have anything to put there, we'll delete the file.

## 5.2 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 configuration files so that when you upgrade a package, you'll be prompted whether you want to keep your old configuration or not.

Since version 3, `dh_installdeb` will automatically flag 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 there is (and should be conffiles) 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 mark them as 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 file in order to work at all, also consider not marking the file as a conffile.

You can handle example configuration files from the 'maintainer scripts', for more information see 'postinst.ex, preinst.ex, postrm.ex, prerm.ex' on page 29.

## 5.3 Cron files

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

- `cron.hourly` - Installed as `/etc/cron.hourly/package`: run once an hour every hour.
- `cron.daily` - Installed as `/etc/cron.daily/package`: run once a day, usually early morning.
- `cron.weekly` - Installed as `/etc/cron.weekly/package`: run once a week, usually early Sunday morning.
- `cron.monthly` - Installed as `/etc/cron.monthly/package`: run once a month, usually early morning on the first of the month.
- `cron.d` - Installed as `/etc/cron.d/package`: for any other time

For the named files, the format of them is a shell script. The different one is `package.cron.d` which follows the format of `crontab(5)`.

Note that this doesn't include log rotation; for that, see `dh_installogrotate(1)` and `logrotate(8)`.

## 5.4 dirs

This file specifies the directories which we need but the normal installation procedure (`make install`) somehow doesn't create. This generally means there is a problem with the Makefile.

It is best to first try to run the installation first and only use this if you run into trouble. There is no preceding slash on the directory names. `dh_install` when it uses the 'files' file doesn't need the directories created first.

## 5.5 docs

This file specifies the file names of documentation files we can have `dh_installdocs` 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, I also included some other stuff:

```
BUGS
CONFIG-CHANGES
CREDITS
ONEWS
README
README.gtkrc
TODO
```

## 5.6 emacsen-\*.ex

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_installemacsen(1)`.

If you don't need these, remove them.

## 5.7 files

If there are files that need to be installed into your package but your standard make install won't do it, you put the filenames and destinations into this file. You should first check there is not a more specific tool to use before; for example documents should be in the docs file and not this one.

This 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 where a binary is forgotten to be installed, the files file would look like:

```
src/foo/mybin usr/bin
```

This will mean when this package is installed, there will be a binary file /usr/bin/mybin.

## 5.8 init.d and package.default

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

This is a fairly generic skeleton file for an /etc/init.d/ script, so you'll likely have to edit it, a lot. It gets installed into the temporary directory by `dh_installinit(1)`.

package.default will be installed into etc/default/package. This file sets defaults that are sourced by the init script. Most times this default file is used to disable running a daemon, set some default flags or timeouts. If your init script has certain 'settable' features you want to install these into the default file, not the init script.

If your upstream package has an init file you can either use it or not. If you don't use their init.d script then create a new one in debian/init.d However if the upstream init script looks fine and installs in the right place you still need to setup 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.

## 5.9 manpage.1.ex, manpage.sgml.ex, manpage.xml.ex

Your program(s) should have a manual page. If they don't, each of these files is a template that you can fill out.

**manpage.1.ex** Manual pages are normally written in `nroff(1)`. The `manpage.1.ex` example 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 include 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:

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

So gentoo’s man page should be called `gentoo.1`. There was no `gentoo.1` man page in the original source so I wrote it using information from the example and from upstream docs.

**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:

- install the `docbook-to-man` package
- add `docbook-to-man` to the `Build-Depends` line in the `control` file
- remove the comment from the `docbook-to-man` invocation in the ‘build’ target of your `rules` file

And remember to rename the file to something like `gentoo.sgml!`

**manpage.xml.ex** If you prefer XML over SGML, you can use the `manpage.xml.ex` template. If you do this, you have two choices:

- 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 a rule to the ‘build’ target of your `rules` file:

```
xsltproc --nonet \
  --param make.year.ranges 1 \
  --param make.single.year.ranges 1 \
```

```
--param man.charmap.use.subset 0 \
-o debian/ \
/usr/share/xml/docbook/stylesheet/nwalsh/manpages/docbook
debian/manpage.xml
```

Alternatively:

- install the `docbook2x` package
- add the `docbook2x` package to the `Build-Depends` line in the `control` file
- add a rule to the 'build' target of your `rules` file:

```
docbook2-man debian/manpage.xml
```

Rename the source file(s) to something like `gentoo.1.xml` or so and check the package documentations for `stylesheet` parameters and output options.

## 5.10 menu.ex

X Window System users usually have a window manager with a menu that can be customized to launch programs. If they have installed the Debian `menu` package, a set of menus for every program on the system will be created for them.

Here's the default `menu.ex` file that `dh_make` created:

```
?package (gentoo) : needs="X11|text|vc|wm" section="Applications/see-menu-manu
title="gentoo" command="/usr/bin/gentoo"
```

The first field after the colon character is "needs", and it specifies what kind of interface the program needs. Change this to one of the listed alternatives, e.g. "text" or "X11".

The next is "section", where the menu and submenu the entry should appear in. The current list of sections is at: `/usr/share/doc/debian-policy/menu-policy.html/ch2.html#s2.1`

The "title" field is the name of the program. You can start this one in uppercase if you like. Just keep it short.

Finally, the "command" field is the command that runs the program.

Now we'll change the menu entry to this:

```
?package (gentoo) : needs="X11" section="Applications/Tools" title="Gentoo" c
```

You can also add other fields like "longtitle", "icon", "hints" etc. See `menufile(5)`, `update-menus(1)` and `/usr/share/doc/debian-policy/menu-policy.html/` for more information.

## 5.11 package.doc-base.EX

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

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

This is how gentoo's doc-base file `gentoo.doc-base.EX` 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
Index: /usr/share/doc/gentoo/html/index.html
Files: /usr/share/doc/gentoo/html/*.html
```

For information on the file format, see `install-docs(8)` and the doc-base manual, in `/usr/share/doc/doc-base/doc-base.html/.`

For more details on installing additional documentation, look in 'Installation in a subdirectory' on page 9.

## 5.12 postinst.ex, preinst.ex, postrm.ex, prerm.ex

These files are called maintainer scripts. They are scripts which are put in the control area of the package and run by `dpkg` when your package is installed, upgraded or removed.

For now, you should try to avoid any manual editing of maintainer scripts if you possibly can because they tend to get complex. For more information look in the Policy Manual, chapter 6, and take a look at these example files provided by `dh_make`.

## 5.13 watch.ex

This file is used to configure the `uscan(1)` and `uupdate(1)` programs (in the `devscripts` package). These are used to watch the site you got the original source from.

Here's what I put in it:

```
# watch control file for uscan
```

```
version=3
#
http://ftp.obsession.se/gentoo/ gentoo-(.*)\.tar\.gz
```

Hint: connect to the Internet, and try running “uscan” in the program directory once you create the file. And read the manuals! :)

## 5.14 source/format

In the file `debian/source/format` there should be a single line describing the type of format your source file will be. It should say either ‘3.0 (native)’ for Debian native packages or ‘3.0 (quilt)’ for everything else. Using this source format means how you use the quilt patch system changes slightly.

## Chapter 6

# Building the package

We should now be ready to build the package.

### 6.1 Complete rebuild

Enter the program's main directory and then issue this command:

```
dpkg-buildpackage -rfakeroot
```

This will do everything for you. It will:

- clean the source tree (debian/rules clean), using `fakeroot`
- build the source package (`dpkg-source -b`)
- build the program (debian/rules build)
- build the binary package (debian/rules binary), using `fakeroot`
- sign the source `.dsc` file, using `gnupg`
- create and sign the upload `.changes` file, using `dpkg-genchanges` and `gnupg`

The only input that will be required of you is your GPG key secret pass phrase, twice.

After all this is done, you will see the following files in the directory above (`~/gentoo/`):

- `gentoo_0.9.12.orig.tar.gz`  
This is the original source code tarball, merely renamed to the above so that it adheres to the Debian standard. Note that this was created using the `'-f'` option to `dh_make` when we initially ran it.

- *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)`. This file is GPG signed, so that people can be sure that it’s really yours.

- *gentoo\_0.9.12-1.diff.gz*

This compressed file contains each and every addition you made to the original source code, in the form known as “unified diff”. It is made and used by `dpkg-source(1)`. Warning: if you don’t name the original tarball `packagename_version.orig.tar.gz`, `dpkg-source` will fail to generate the `.diff.gz` file properly!

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`.

- *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, and it is used by the Debian FTP archive maintenance programs to install the binary and source packages in it. It is partly generated from the ‘changelog’ file and the `.dsc` file. This file is GPG signed, so that people can be sure that it’s really yours.

As you keep working on the package, 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` mailing list.

The long strings of numbers in the `.dsc` and `.changes` files are MD5 checksums for the files mentioned. A person downloading your files can test them with `md5sum(1)` and if the numbers don’t match, they’ll know the file is corrupt or has been tampered with.

## 6.2 Quick rebuild

With a large package, you may not want to rebuild from scratch every time while you tune a detail in `debian/rules`. For testing purposes, you can make a `.deb` file without rebuilding the upstream sources like this:

```
fakeroot debian/rules binary
```

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

### 6.3 The `debuild` command

You can automate package build process further with `debuild` command. See `debuild(1)`.

Customization of the `debuild` command can be done through `/etc/devscripts.conf` or `~/.devscripts`. I would suggest at least following items:

```
DEBSIGN_KEYID="Your_GPG_keyID"
DEBUILD_DPKG_BUILDPACKAGE_OPTS="-i -ICVS -I.svn"
```

With these, you can build package always with your GPG key and avoid including undesired components. (This is good for sponsoring too.) For example, cleaning source and rebuilding package from a user account is as simple as:

```
debuild clean
debuild
```

### 6.4 The `dpatch` and `quilt` systems

The simple use of `dh_make` and `dpkg-buildpackage` commands will create a single large `diff.gz` file which contains package maintenance files in `debian/` and patch files to the source. Such a package is a bit cumbersome to inspect and understand for each source tree modification later. This is not so nice.<sup>1</sup>

Several methods for the patch set maintenance have been proposed and are in use with Debian packages. The `dpatch` and `quilt` systems are two of the simplest of such patch maintenance systems proposed. Other ones are `db`s, `cdbs`, etc.

A package which is packaged properly with the `dpatch` or `quilt` systems has modifications to the source clearly documented as a set of `-p1` patch files with header in `debian/patches/` and the source tree is untouched outside of `debian/` directory. If you are asking a sponsor to upload your package, this kind of clear separation and documentation of your changes are very important to expedite the package review by your sponsor. The usage method of `dpatch` and `quilt` is explained in `dpatch(1)`, `dpatch-edit-patch(1)` and `quilt(1)`. Both programs provide convenience files to include in `debian/rules`: `/usr/share/dpatch/dpatch.make` and `/usr/share/quilt/quilt.make`.

When someone (including yourself) provides you with a patch to the source later, then the package modification is quite simple:

- Edit patch to make it a `-p1` patch to the source tree.
- In the case of `dpatch`, add header using `'dpatch patch-template'` command.
- Drop it into `debian/patches`

---

<sup>1</sup>If you are not yet Debian Developer and asking your sponsor to upload your package after his package review, you should make package as easy as possible for him to review.

- Add the patch filenames to `debian/patches/00list` (for `dpatch`) or `debian/patches/series` (for `quilt`).

Also, `dpatch` has a capability to make patches architecture dependent using CPP macro.

## 6.5 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. If package version is not at `-0` or `-1` Debian revision, you must provide `dpkg-buildpackage` command with the `"-sa"` option. On the other hand, the `"-sd"` option will force to exclude the original `orig.tar.gz` source.

---

## Chapter 7

# Checking the package for errors

### 7.1 The `lintian` package

Run `lintian(1)` on your `.changes` file; these programs will check for many common packaging errors. The commands are:

```
lintian -i gentoo_0.9.12-1_i386.changes
```

Of course, replace the filename with the name of the `.changes` file generated for your package. If it appears that there are some errors (lines beginning with E:), read the explanation (the N: lines), correct mistakes, and rebuild as described in ‘Complete rebuild’ on page 31. If there are lines that begin with W:; those are warnings, so tune the package or verify that the warnings are spurious (and make Lintian overrides; see the documentation for details).

Note that you can build the package with `dpkg-buildpackage` and run `lintian` all in one command with `debuild(1)`.

### 7.2 The `mc` command

You can unpack the contents of `*.deb` package with `dpkg-deb(1)` command. You can list the contents of a generated Debian package with `debc(1)`.

This 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 `*.diff.gz` and `*.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 that.

Tips: `'zgrep ^+++ ../gentoo_0.9.12-1.diff.gz'` will give you a list of your changes/additions to the source files, and `'dpkg-deb -c gentoo_0.9.12-1_i386.deb'` or `'debc gentoo_0.9.12-1_i386.changes'` will list the files in the binary package.

### 7.3 The `debdiff` command

You can compare file lists in two binary Debian packages with `debdiff(1)` command. This is useful for verifying that no files have been unintentionally misplaced or removed, and no other inadvertent changes were made when updating packages. You can check group of `*.deb` files simply by `'debdiff old-package.change new-package.change'`.

### 7.4 The `interdiff` command

You can compare two `diff.gz` files with `interdiff(1)` command. This is useful for verifying that no inadvertent changes were made to the source by the maintainer when updating packages. Run `'interdiff -z old-package.diff.gz new-package.diff.gz'`.

### 7.5 The `debi` command

Install the package to test it yourself, e.g. using the `debi(1)` command as root. Try to install and run it on machines other than your own and watch closely for any warnings or errors both during the installation and while the program is being run.

### 7.6 The `pbuilder` package

For a clean room (chroot) build environment to verify the build dependencies, the `pbuilder` package is very useful. This ensures a clean build from source under the auto-builder for different architectures and avoids the severity serious FTBFS (Fails To Build From Source) bug which is always in the RC (release critical) category. See <http://buildd.debian.org/> for more on the Debian package auto-builder.

The most basic use of the `pbuilder` package is the direct invocation of `pbuilder` command from root. For example, issue the following commands in the directory where `.orig.tar.gz`, `.diff.gz`, and `.dsc` exist to build a package.

```
root # pbuilder create # if second time, pbuilder update
root # pbuilder build foo.dsc
```

The newly built packages will be located in `/var/cache/pbuilder/result/` with root ownership.

The `pdebuild` command helps you to use `pbuilder` package functions from the normal user account. From the root of the source tree while having `orig.tar.gz` file in its parent directory, you issue the following commands:

```
$ sudo pbuilder create # if second time, sudo pbuilder update
$ pdebuild
```

The newly built packages will be located in `/var/cache/pbuilder/result/` with non-root ownership.<sup>1</sup>

If you want to add an additional apt source to be used by the `pbuilder` package, you set `OTHERMIRROR` in `~/ .pbuilderrc` or `/etc/pbuilderrc` and run (for `sarge`)

```
$ sudo pbuilder update --distribution sarge --override-config
```

The use of `--override-config` is needed to update apt source within chroot environment.

See <http://www.netfort.gr.jp/~dancer/software/pbuilder.html>, `pdebuild(1)`, `pbuilderrc(5)`, and `pbuilder(8)`.

---

<sup>1</sup>Currently, I would suggest customizing your system by setting `/var/cache/pbuilder/result/` directory writable by the user and setting `~/ .pbuilderrc` or `/etc/pbuilderrc` to include

```
AUTO_DEBSIGN=yes
```

This will allow you to sign generated packages with your secret GPG key under `~/ .gnupg/`. Since the `pbuilder` package is still evolving, you have to check the actual configuration situation by consulting the latest official documentation.



---

## Chapter 8

# Uploading the package

Now that you have tested your new package thoroughly, you will be ready to start the Debian new maintainer application process, as described at <http://www.debian.org/devel/join/newmaint>

### 8.1 Uploading to the Debian archive

Once you become an official developer, you'll need to upload the package to the Debian archive. You can do this manually, but it's easier to use the provided 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. Put something like this in the file:

```
package config;

$default_host = "anonymous-ftp-master";

$config{'anonymous-ftp-master'} = {
    fqdn => "ftp-master.debian.org",
    method => "ftp",
    incoming => "/pub/UploadQueue/",
    # files pass on to dinstall on ftp-master which sends emails itself
    dinstall_runs => 1,
};

1;
```

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

The `$default_host` option is the trickiest one – it 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, faster one. For more information about the upload queues, read the Developers’ Reference, section “Uploading a package”, at `/usr/share/doc/developers-reference/pkgs.html#upload`

Then connect to your Internet provider, and issue this command:

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

`dupload` checks that the MD5 checksums of the files match those from the `.changes` file, so it will warn you to rebuild it as described in ‘Complete rebuild’ on page 31 so it can properly upload.

If you encounter an upload problem at <ftp://ftp-master.debian.org/pub/UploadQueue/>, you can fix this by manually uploading `gnupg` signed `*.commands` file to <ftp://ftp-master.debian.org/pub/UploadQueue/> with `ftp`.<sup>1</sup> For example, use `hello.commands`:

```
-----BEGIN PGP SIGNED MESSAGE-----

Uploader: Roman Hodek <Roman.Hodek@informatik.uni-erlangen.de>
Commands:
  rm hello_1.0-1_i386.deb
  mv hello_1.0-1.dsx hello_1.0-1.dsc

-----BEGIN PGP SIGNATURE-----
Version: 2.6.3ia

iQCVAwUBNFiQSXVhJ0HiWnvJAQG58AP+IDJVeSWmDvzMUpHScg1EK0mvChgnuD7h
BRiVQubXkB2DphLJW5UUSRnjw1iuFcYwH/lFpNp17XP95LkLX3iFza9qItw4k2/q
tvylZkmIA9jxCyv/YB6zZCbHmbvUnL473eLRoxlnYZd3JFaCZMJ86B0Ph4GFNPaf
Z4jxNrgh7Bc=
=pH94
-----END PGP SIGNATURE-----
```

## 8.2 Uploading to a private archive

If you want to create a personal package archive at `URL="http://people.debian.org/~account_name"` as a developer with simple invocation of `dupload -t target_name`, you should add the following to `/etc/dupload.conf` file:

<sup>1</sup>See <ftp://ftp-master.debian.org/pub/UploadQueue/README>. Alternatively, you may use `dcut` command from the `dput` package.

```
# Developer account
$cfg{'target_name'} = {
    fqdn => "people.debian.org",
    method => "scpb",
    incoming => "/home/account_name/public_html/package/",
    # I do not need to announce
    dinstall_runs => 1,
};
$cfg{'target_name'}{preupload}{'changes'} = "
    echo 'mkdir -p public_html/package' | ssh people.debian.org 2>/dev/n
    echo 'Package directory created!';

$cfg{'target_name'}{postupload}{'changes'} = "
    echo 'cd public_html/package ;
    dpkg-scanpackages . /dev/null >Packages || true ;
    dpkg-scansources . /dev/null >Sources || true ;
    gzip -c Packages >Packages.gz ;
    gzip -c Sources >Sources.gz ' | ssh people.debian.org 2>/dev/null ;
    echo 'Package archive created!';
```

Here, the APT archive is built with a quick and dirty remote shell execution with SSH. The override files required by `dpkg-scanpackages` and `dpkg-scansources` are given as `/dev/null`. This technique can be used by a non Debian Developer to host his packages on his personal web site. Alternatively you can use `apt-ftparchive` or other scripts to create an APT archive.



## Chapter 9

# Updating the package

### 9.1 New Debian revision

Let's say that a bug report was filed against your package, #54321, and it describes a problem that you can solve. To create a new Debian revision of the package, you need to:

- Correct the problem in the package source, of course.
- 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.

Tip: how to easily get the date in required format? Use `'822-date'`, or `'date -R'`.

- Include a short description of the bug and the solution in the changelog entry, followed by this: "Closes: #54321". That way, the bug report will be automatically closed by the archive maintenance software the moment your package gets accepted in the Debian archive.
- Repeat what you did in 'Complete rebuild' on page 31, 'Checking the package for errors' on page 35, and 'Uploading the package' on page 39. 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.

### 9.2 New upstream release (basic)

Now let's consider a different, slightly more complicated situation - a new upstream version was released, and of course you want it packaged. You need to do the following:

- Download the new sources and put the tarball (e.g. named `'gentoo-0.9.13.tar.gz'`) in the directory above the old source tree (e.g. `~/gentoo/`).

- Enter the old source directory, and run:

```
uupdate -u gentoo-0.9.13.tar.gz
```

Of course, replace this file name with the name of your program's new source archive. `uupdate(1)` will properly rename that tarball, try to apply all the changes from your previous `.diff.gz` file, and update the new `debian/changelog` file.

- Change directory to `'.. /gentoo-0.9.13'`, the new package source tree, and repeat what you did in 'Complete rebuild' on page 31, 'Checking the package for errors' on page 35, and 'Uploading the package' on page 39.

Note that if you set up a `'debian/watch'` file as described in 'watch.ex' on page 29, you can run `uscan(1)` to automagically look for revised sources, download them, and run `uupdate`.

### 9.3 New upstream release (realistic)

When preparing packages for the Debian archive, you must check the resulting packages in detail. Here is a more realistic example of this procedure.

#### 1 Verify changes in upstream source

- Read the upstream `changelog`, `NEWS`, and whatever other documentation they may have released with the new version.
- Do a `'diff -urN'` between the old and new upstream sources to try to get a feel for the scope of the changes, where work is actively being done (and thus where new bugs may appear), and also keep an eye out for anything suspicious.

#### 2 Port the old Debian packaging to the new version.

- Unpack the source tarball and rename the root of the source tree as `<packagename>-<upstream_version>/` and `'cd'` into this directory.
- Copy the source tarball in the parent directory and rename it as `<packagename>_<upstream_version>.orig.tar.gz`.
- Apply the same kind of modification to the new source tree as the old source tree.

Possible methods are:

- `'zcat /path/to/<packagename>_<old-version>.diff.gz | patch -p1'` command,
- `'uupdate'` command,
- `'svn merge'` command if you manage the source with Subversion repository, or
- simply copying `debian/` directory from the old source tree if it was packaged with `dpatch` or `quilt`.
- Preserve old `changelog` entries (sounds obvious, but there have been incidents...)
- The new package version is the upstream release version appended with a `-1` Debian revision number, e.g., `'0.9.13-1'`.
- Add `changelog` record entry with "New upstream release" for this new version at the top of `debian/changelog`. For example `'dch -v 0.9.13-1'`.

- Describe concisely the changes *in* the new upstream release that fix reported bugs and close those bugs in the changelog.
  - Describe concisely the changes *to* the new upstream release by the maintainer that fix reported bugs and close those bugs in the changelog.
  - If the patch/merge did not apply cleanly, inspect the situation to determine what failed (clues are left in `.rej` files). Most often the problem is that a patch you applied to the source was integrated upstream, and thus the patch is no longer relevant.
  - Upgrades to the new version should be silent and nonintrusive (existing users should not notice the upgrade except by discovering that old bugs have been fixed and there perhaps are new features).<sup>1</sup>
  - If you need to add erased template files for any reason, you may run `dh_make` again in the same, already “debianized”, directory with `-o` option. Then edit it properly.
  - 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 step 1) then update the `debian/rules` and `debian/control` build dependencies if necessary.
- 3 Build the new package as described in ‘The `debuild` command’ on page 33 or ‘The `pbuilder` package’ on page 36. Use of `pbuilder` is desirable.
  - 4 Verify new packages are built correctly.
    - Perform ‘Checking the package for errors’ on page 35.
    - Perform ‘Verifying package upgrades’ on the next page.
    - Check again to see if any bugs have been fixed that are currently open in the Debian Bug Tracking System (BTS) (<http://www.debian.org/Bugs/>).
    - Check the contents of the `.changes` file to make sure you are uploading to the correct distribution, the proper bugs closures are listed in the `Closes:` field, the `Maintainer:` and `Changed-By:` fields match, the file is GPG-signed, etc.
  - 5 If any changes were made to correct anything in the packaging along the way, go back to the step 2 until satisfied.
  - 6 If your upload needs to be sponsored, be sure to note any special options required when building the package (like `'dpkg-buildpackage -sa -v ...'`) and be sure to inform your sponsor so he or she builds it correctly.
  - 7 If you are uploading yourself, perform ‘Uploading the package’ on page 39.

---

<sup>1</sup>Please make your package properly updates the config file upon upgrades using well designed `postinst` etc., so that it **doesn’t** do things not wanted by the user! These are the enhancements that explain **why** people choose Debian. When the upgrade is necessarily intrusive (eg., config files scattered through various home directories with totally different structure), you may consider to set package to the safe default (e.g., disabled service) and provide proper documentations required by the policy (`README.Debian` and `NEWS.Debian`) as the last resort. But don’t bother with the `debconf` note.

## 9.4 The `orig.tar.gz` file

If you try to build packages only from the new source tree with `debian/` directory without the `orig.tar.gz` file in its parent directory, you will end up unintentionally creating a native source package, which comes without the `diff.gz` file. This type of packaging is only appropriate for the debian-specific packages, which will never be useful in another distribution.<sup>2</sup>

In order to obtain a non-native source package which consists of both the `orig.tar.gz` file and the `diff.gz` file, you must manually copy the upstream tarball to the parent directory with its file name changed into `<packagename>_<upstream_version>.orig.tar.gz` as it was done by `dh_make` command in ‘Initial “debianization”’ on page 7.

## 9.5 The `cvs-buildpackage` command and similes

You should consider using a source code management system to manage packaging activity. There are several wrapper scripts which are customized to be used with the most popular ones.

- CVS
  - `cvs-buildpackage`
- Subversion
  - `svn-buildpackage`
- Git (git-core)
  - `git-buildpackage`

These commands also automate the packaging of new upstream releases.

## 9.6 Verifying package upgrades

When you build a new version of the package, you should do the following to verify that the package can be safely upgraded:

- upgrade from the previous version
- downgrade back again and then remove it,
- install the new package
- remove it and then reinstall it again,
- purge it.

---

<sup>2</sup>Some people argue that, even for Debian specific packages, it is still better practice to package the contents of the `debian/` directory residing in the `diff.gz` file, rather than in the `orig.tar.gz` file.

If the package makes use of non-trivial pre/post/inst/rm scripts, be sure to test the upgrade paths of those.

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.



## Chapter 10

# Where to ask for help

Before you decide to ask your question in some public place, please just RTFM. That includes documentation in `/usr/share/doc/dpkg`, `/usr/share/doc/debian`, `/usr/share/doc/autotools-dev/README.Debian.gz`, `/usr/share/doc/package/*` files and the `man/info` pages for all the programs mentioned in this document. See all the information at <http://nm.debian.org/> and [http://people.debian.org/~mpalmer/debian-mentors\\_FAQ.html](http://people.debian.org/~mpalmer/debian-mentors_FAQ.html).

If you have questions about packaging that you couldn't find answers to in the documentation, you can ask them on the Debian Mentors' mailing list at `<debian-mentors@lists.debian.org>`. The more experienced Debian developers will gladly help you, but do read at least some of the documentation before asking a question!

See <http://lists.debian.org/debian-mentors/> for more information about this mailing list.

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 Developers' Reference, chapter "Handling Bugs", at `/usr/share/doc/developers-reference/pkgs.html#bug-handling`

If you still have questions, ask on the Debian Developers' mailing list at `<debian-devel@lists.debian.org>`. See <http://lists.debian.org/debian-devel/> for more information about this mailing list.

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 (once again, read the *real documentation* for details). Good luck!



## Appendix A

# Examples

Here we package the upstream tarball *gentoo-1.0.2.tar.gz* and uploading all the packages to the *nm\_target*.

### A.1 Simple packaging example

```
$ mkdir -p /path/to # new empty directory
$ cd /path/to
$ tar -xvzf /path/from/gentoo-1.0.2.tar.gz # get source
$ cd gentoo-1.0.2
$ dh_make -e name@domain.dom -f /path/from/gentoo-1.0.2.tar.gz
... Answer prompts.
... Fix source tree
... If it is a script package, set debian/control to "Architecture: all"
... Do not erase ../gentoo_1.0.2.orig.tar.gz
$ debuild
... Make sure no warning happens.
$ cd ..
$ dupload -t nm_target gentoo_1.0.2-1_i386.changes
```

### A.2 Packaging example with the *dpatch* and the *pbuilder*

```
$ mkdir -p /path/to # new empty directory
$ cd /path/to
$ tar -xvzf /path/from/gentoo-1.0.2.tar.gz
$ cp -a gentoo-1.0.2 gentoo-1.0.2-orig
$ cd gentoo-1.0.2
$ dh_make -e name@domain.dom -f /path/from/gentoo-1.0.2.tar.gz
... Answer prompts.
```

Here part of `debian/rules` originally looks like:

```
configure: configure-stamp
configure-stamp:
    dh_testdir
    # Add here commands to configure the package.
    touch configure-stamp
build: build-stamp
build-stamp: configure-stamp
    dh_testdir
    # Add here commands to compile the package.
    $(MAKE)
    #docbook-to-man debian/gentoo.sgml > gentoo.1
    touch $@
clean:
    dh_testdir
    dh_testroot
    rm -f build-stamp configure-stamp
    # Add here commands to clean up after the build process.
    -$(MAKE) clean
    dh_clean
```

You change this part of `debian/rules` to the following by the editor to use `dpatch` and add `dpatch` to the `Build-Depends`: line in the `debian/control` file:

```
configure: configure-stamp
configure-stamp: patch
    dh_testdir
    # Add here commands to configure the package.
    touch configure-stamp
build: build-stamp
build-stamp: configure-stamp
    dh_testdir
    # Add here commands to compile the package.
    $(MAKE)
    #docbook-to-man debian/gentoo.sgml > gentoo.1
    touch $@
clean: clean-patched unpatch
clean-patched:
    dh_testdir
    dh_testroot
    rm -f build-stamp configure-stamp
    # Add here commands to clean up after the build process.
    -$(MAKE) clean
    dh_clean
```

```
patch: patch-stamp
patch-stamp:
    dpatch apply-all
    dpatch call-all -a=pkg-info >patch-stamp
unpatch:
    dpatch deapply-all
    rm -rf patch-stamp debian/patched
```

Now you are ready to repack the source tree with dpatch system with the help of dpatch-edit-patch.

```
$ dpatch-edit-patch patch 10_firstpatch
... Fix source tree by editor
$ exit 0
... Try building packages with "debuild -us -uc"
... Clean source with "debuild clean"
... Repeat dpatch-edit-patch until making source buildable.
$ sudo pbuilder update
$ pdebuild
$ cd /var/cache/pbuilder/result/
$ dupload -t nm_target gentoo_1.0.2-1_i386.changes
```