Debian New Maintainers' Guide

Josip Rodin <jrodin@jagor.srce.hr>

version 1.0.2, 10 June 2001.

Contents

Chapter 1

Getting started "The Right Way"

This document will try to describe building of a Debian GNU/Linux package to the common Debian user (and wannabe developer) in common language, and well covered with working examples. There is an old Roman saying, *Longum iter est per preaecepta, 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.

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 document has been written while the 2.1 'slink' distribution was the officially stable release of Debian, and the 2.2 'potato' was being created, thus the packages named here are mostly those from 'potato'.

The following packages come with standard installation of Debian 2.1, so you probably have them (and additional packages they depend on) already. Still, you should check with 'dpkg –s <package>'.

- *binutils* – these programs used to assemble and link object files – the stuff programs are made of. (see 'info binutils')
- *cpp* – the C preprocessor. (see `cpp(1)`)

- *cpio* – this is an archiver like tar or zip. (see `cpio(1)`)
- *dpkg–dev* – this package contains the tools needed to unpack, build and upload Debian source packages. (see `dpkg-source(1)`)
- *file* – this handy program can determine what type a file is. (see `file(1)`)
- *gcc* – the GNU C compiler. Most Linux programs are written in C. However, if your program is written in some other programming language, like C++, Fortran, Pascal, you should get g++, g77, or gpc, respectively. (see `gcc(1)`, `g++(1)`, `g77(1)`, `gpc(1)`)
- *libc6–dev* – the C libraries and header files gcc needs to link with to create object files. Although some programs still recommend and/or use libc5, you are encouraged to use the newer version (libc6). (see ‘info libc‘)
- *make* – usually creation of a program takes several steps. Rather than having to type out the same commands over and over again, you can use this program to automate the process, creating ‘Makefile’s. Some programs also use imake and xmkmf, programs used to generate Makefiles from sets of macro functions. Many newer programs use configure scripts and Makefiles with help of programs like autoconf and automake, so you might need these, too. (see ‘info make‘, `imake(1)`, `xmkmf(1)`, `autoconf(1)`, `automake(1)`)
- *patch* – this very useful utility will take file containing a difference listing (produced by the diff program) and apply it to the original file, producing a patched version. (see `patch(1)`)
- *perl5* – Perl is one of the most used interpreted scripting languages on today’s un*x systems, often referred to as "Unix’s Swiss Army Chainsaw". (see `perl(1)`)

From the ‘devel’ section of the distribution you’ll probably need to install these yourself:

- *dh–make* and *debhelper* – dh–make is necessary to create the skeleton of our example package, and it will use some the debhelper tools for creating packages. They are not essential for creation of packages, but it is **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 contain 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)`)
- *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)

Finally, these *very important* packages are in the doc section of the distribution:

- *debian–policy* – includes the structure and contents of the archive, several OS design issues, the Filesystem Hierarchy Standard, and the most important thing (for you) 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, when and where to upload etc. (see /usr/share/doc/developers–reference/developers–reference.html/index.html)

You will also need the encryption package, either PGP (the pgp–* packages) or GPG (the gnupg package), to digitally *sign* your package. 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). However, due to a rather wacky U.S. export law, you cannot simply download this from your nearest Debian FTP site. But Debian does provide these packages through a site that is not physically located in the US, called non–US.debian.org (ftp://non–us.debian.org/debian–non–US/). Your FTP site will have a file called README.non–US, which will tell you how to find a closer mirror of that site.

The short descriptions that are given above only serve to introduce you to what each package does. Before continuing please thoroughly read how to 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. Note: *debmake* is a package that contains some programs that function similar to dh–make, but its specific use is **not** covered in this document. Refer to the Debmake manual (`http://www.debian.org/~jaldhar/`) for more information.

1.2 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.

After you build your package (or while doing that), you will have to become an official Debian maintainer if you wish your program to get into the next distribution (if the program is useful, why not?). That process is explained in Developer's Reference. Please read it.

Chapter 2

First steps

While the documentation at the Developers corner (`http://www.debian.org/devel/`) is not so clear about where and how new maintainers should start their work, 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.

2.1 Choose your program.

You have probably chosen the package you want to build, but here are some pointers for you as the uninitiated:

- check if the package is in the distribution already. If you use the 'stable' distribution, maybe it's best that you go to the package search page (`http://www.debian.org/distrib/packages.html`). If you use **current** 'unstable' distribution, check it out with these commands:

  ```
  dpkg -s program
  dpkg -l '*program*'
  ```

- consult the WNPP page (`http://www.debian.org/devel/wnpp/`) and debian–devel mailing list archives to see if anyone else is building that same package. If so, contact the current maintainer if you feel you need to. If not – find another interesting program that nobody maintains.
- program **must** have a license, if possible free as according to the Debian Free Software Guidelines (`http://www.debian.org/social_contract.html#guidelines`). If it doesn't conform to some of these rules, it still can be included in 'contrib' or 'non–free' sections of Debian. If you are unsure about where should it go, ask on <`debian-legal@lists.debian.org`>.
- 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.

- program should result in binary executable form, don't try libraries yet.
- it should be well documented, or at least understandable (to anyone).
- 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 the least, you must know that it works, and have it tried 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 most experienced developers consult debian–devel 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 Linux 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 debian–mentors if you're not sure how to unpack it correctly (hint: issue 'file archive.extension').

As an example, I'll use program called 'gentoo', an X11 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 exist 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 type ‘make uninstall‘ to remove all the installed files, and ‘make clean‘ (or better ‘make distclean‘) to clean up the build directory.

2.3 Stuff prior to ‘dh_make’

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, prepended with a “0.0.” (just in case upstream people one day decide to release a nice version like 1.0). So, if the release or snapshot date was 19th of December, 1998, you can use the version string of 0.0.19981219. 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 Running ‘dh_make’

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 with pressing <enter>. As a new maintainer you are discouraged from creating packages with multiple binary packages, or libraries, as explained earlier. It’s not that hard, really, 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 30

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 19.

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 on 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.

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. You have to make sure that they do it correctly, but you have to make them install themselves in a temporary subdirectory that will be created under your debian/ directory, usually called `debian/tmp`, from which the maintainer tools will build a working .deb package. Everything that is contained in this directory will be installed on user's system when he installs your package, the only difference is that dpkg will be installing the files in the root directory.

Basically, you need to make the program install in debian/tmp, but behave correctly when placed in the root directory, i.e. when installed from the .deb package. With programs using GNU autoconf, this will be quite easy, because dh_make will set up commands for doing that 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'? Note: if you change this,
# gentoo will not find the icons as it starts up. You're going to
# have to alter gentoo's icon path (in the config window, "Paths"
# tab) to get it work.
ICONS   = /usr/local/lib/gentoo/
```

Before all that, you should insert a new two lines that say:

```
# Edited for Debian GNU/Linux.
DESTDIR =
```

because the build process requires it (will be explained later, in "rules' file' on page 19).

Then the Makefile mentions the location of the final binary. Just change that to this:

```
# Where to put binary on 'make install'?
BIN     = $(DESTDIR)/usr/X11R6/bin
```

But why in that directory, and not some other? Because Debian has defined some rules on where programs are to be installed. That is specified in the Filesystem Hierarchy Standard (see /usr/share/doc/debian–policy/fhs/). So, we should install the binary in /usr/X11R6/bin instead of /usr/local/bin, and the man page (it doesn't exist here, but almost every program has one, so we'll make one later) in /usr/share/man/man1 instead of /usr/local/man/man1.

After that we have a bit harder situation. If you change one line to:

```
ICONS   = $(DESTDIR)/usr/share/gentoo/
```

which conforms to the policy, you will have to edit some real C sources. But where and what to search? You can find this out by issuing:

```
grep -n usr/local/lib *.[ch]
```

(in every subdirectory that contains .c and .h files). Grep will tell you the name of the file and the line in it, when it finds an occurrence. Now edit those files and in those lines replace usr/local/lib with usr/share – and that is it. Just replace usr/local/lib with your location, and be very careful that you don't mess up the rest of the code, if you don't know much about programming in C. :–)

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

```
# ----------------------------------------- Installation

# You're going to have to be root to do this!
install:        gentoo
                install ./gentoo $(BIN)
                install icons $(ICONS)
                install gentoorc-example $(HOME)/.gentoorc
```

After our change it says:

```
# ----------------------------------------- Installation

# You're going to have to be root to do this!
install:        gentoo-target
                install -d $(BIN) $(ICONS) $(DESTDIR)/etc
                install ./gentoo $(BIN)
                install -m644 icons/* $(ICONS)
                install -m644 gentoorc-example $(DESTDIR)/etc/gentoorc
                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 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. Note that you don't have to send the debian/* files upstream, but you should do so with any other patches. And try to be nice to upstream, by making your fixes not specific to Debian or Linux (or even Unix!) prior to sending them.

3.2 Differing libraries

There is one other common problem: libraries are often different from platform to platform. For example, Makefile can contain a reference to a library which doesn't exist on Debian, or even Linux. In that case, we need to change it to a library which does exist in Debian, and serves the same purpose. The best way is to comment out those lines because there may be others who will try to compile on different platforms, so that they can have some hints about what could be the problem.

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

Chapter 4

Required stuff under debian/

There is a new subdirectory under the program's main directory ('gentoo–0.9.12'), it is called 'debian'. There are a number of files in this directory. We will be editing these 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 and dselect will use to manage the package. Here is the control file dh_make creates for us.

```
1  Source: gentoo
2  Section: unknown
3  Priority: optional
4  Maintainer: Josip Rodin <jrodin@jagor.srce.hr>
5  Standards-Version: 3.0.1
6
7  Package: gentoo
8  Architecture: any
9  Depends: ${shlibs:Depends}
10 Description: <insert up to 60 chars description>
11  <insert long description, indented with spaces>
```

(I've added the line numbers.)

Lines 1–5 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 this 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 specific programs, and many more.

Let's change it then to x11.

Line 3 describes how important it is that the user install this package. Section and priority are actually only used by dselect when it sorts packages and selects defaults, and they can (and most probably will be) overridden by our FTP maintainers. See the Policy manual for guidance on what to set these fields to.

As it is a normal priority package, we'll leave it as optional.

Line 4 is the name and email address of the maintainer.

Line 5 is the version of the Debian Policy standards this package follows (two major versions of the installed debian–policy package).

If some non–standard compiler or other tool is needed to build your package, you should add here a 'Build–Depends' line and list the required packages. For more information on this, read the Packaging Manual (section 8.7) and documentation of the 'build–essential' package.

Line 7 is the name of the binary package.

Line 8 describes the CPU architecture the binary package can be compiled for. We can leave this as "any" because `dpkg-gencontrol(1)` will fill in the appropriate value for any machine this package gets compiled on (see the Developer's Reference for explanation on what porting packages is). 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 19 about using 'binary–indep' rule instead of 'binary–arch' for building the package.

Line 9 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 such as dpkg, dselect or APT (and its front–ends) usually behave the same way when dealing with these relations; if not, it will be explained. (see `dpkg(8)`, `dselect(8)`, `apt(8)`, `console-apt(8)`, `gnome-apt(8)`)

This is what they usually 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:

  Dselect will not install your package unless the packages it recommends are installed. Dpkg and APT should let you do it, though. Use this for packages that are not strictly necessary but are typically used with your program.

- Suggests:

  When a user installs your program, dselect will prompt him to install any package it suggests. Dpkg and APT shouldn't care much. 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 breakage) 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 /usr/share/doc/debian–policy/virtual–package–names–list.text.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 removed before installing yours.

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.

The last feature you need to know about is $(shlibs:Depends). This will be automatically generated by `dh_shlibdeps(1)` and filled in by `dh_gencontrol(1)` with the names of any shared libraries your program uses, such as libc6 or xlib6g, so you don't have to specify them yourself. Having said all that, we can leave line 9 exactly as it is now.

Line 10 is where the list of suggestions go. Here it's only 'file', because gentoo can use some features provided by that program/package.

Line 11 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 "A fully GUI configurable GTK+ file manager".

Line 12 is where the long description goes. This should be a paragraph which gives more detail about the package. Column 1 of each line should be empty. There must be no blank lines, but you can put a . (dot) in a column to simulate that. Also, there must be no more that one blank line after the long description.

Here is the updated control file:

```
Source: gentoo
Section: x11
Priority: optional
Maintainer: Josip Rodin <jrodin@jagor.srce.hr>
Standards-Version: 3.0.1

Package: gentoo
Architecture: any
Depends: ${shlibs:Depends}
Suggests: file
Description: A fully GUI configurable GTK+ file manager
 gentoo is a file manager for Linux written from scratch in pure C. It
 uses the GTK+ toolkit for all of its interface needs. gentoo provides
 100% GUI configurability; no need to edit config files by hand and re-

 start the program. gentoo supports identifying the type of various
 files (using extension, regular expressions, or the 'file' com-
mand),
 and can display files of different types with different col-
ors and icons.
 .
 gentoo borrows some of its look and feel from the classic Amiga file
 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 contents is (section 6.5). Dh_make created a default one, this is what it looks like:

```
This package was debianized by Josip Rodin <jrodin@jagor.srce.hr> on
Wed, 11 Nov 1998 21:02:14 +0100.

It was downloaded from <fill in ftp site>

Upstream Author(s): <put author(s) name and email here>

Copyright:

<Must follow 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. Gentoo is licensed under the GNU General Public License, so we'll change the file to this:

```
This package was debianized by Josip Rodin <jrodin@jagor.srce.hr> on
Wed, 11 Nov 1998 21:02:14 +0100.

It was downloaded from: ftp://ftp.obsession.se/gentoo/

Upstream author: Emil Brink <emil@obsession.se>

This software is copyright (c) 1998-99 by Emil Brink, Obsession
Development.

You are free to distribute this software under the terms of
the GNU General Public License.
On Debian systems, the complete text of the GNU General Public
License can be found in /usr/share/common-licenses/GPL file.
```

(I've added the line numbers.)

4.3 'changelog' file

This is a required file, which has a special format described in the Packaging Manual (section 3.2.3). 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 there any unresolved issues with the package that they should know about instantly. It will be saved as '/usr/share/doc/gentoo/changelog.Debian.gz' in the binary package.

Dh_make creates a default one, this is what it looks like:

```
gentoo (0.9.12-1) unstable; urgency=low

 * Initial Release.

-- Josip Rodin <jrodin@jagor.srce.hr> Wed, 11 Nov 1998 21:02:14 +0100

Local variables:
mode: debian-changelog
End:
```

(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 'experimental' (for now), 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 upstream authors, installed 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)`, `emacs(1)` (lines 7–9 contain mode information for the Emacs editor), or any other 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 <jrodin@jagor.srce.hr> Wed, 11 Nov 1998 21:02:14 +0100
8
9  Local variables:
10  mode: debian-changelog
11 End:
```

(I've added the line numbers.)

When you release a new revision, you must increment the version number. You can do that with just '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'.

New version information is added at the top of the changelog file. This is what the changelog looks like afterwards:

```
1  gentoo (0.9.12-2) unstable; urgency=low
2
3   * Fixed a glitch in the menu file.
4
5  -- Josip Rodin <jrodin@jagor.srce.hr> Wed, 11 Nov 1998 22:15:39 +0100
6
7  gentoo (0.9.12-1) unstable; urgency=low
8
9   * Initial Release.
10  * This is my first Debian package.
11  * Adjusted the Makefile to fix $DESTDIR problems.
12
13 -- Josip Rodin <jrodin@jagor.srce.hr> Wed, 11 Nov 1998 21:02:14 +0100
14
```

```
Local variables:
mode: debian-changelog
End:
```

(I've added the line numbers.)

You can read more about new package versions/revisions later in 'Updating the package' on page 30.

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, since it is executed with 'make –f', but different than the one in the upstream source.

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 that rule depends on. After that, there can be any number of commands (indented with <tab>!), until an empty line is found. There begins another rule. Multiple 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 Packaging Manual.

```
#!/usr/bin/make -f
# Made with the aid of dh_make, by Craig Small
# Sample debian/rules that uses debhelper. GNU copyright 1997 by Joey Hes
# Some lines taken from debmake, by Christoph Lameter.

# Uncomment this to turn on verbose mode.
#export DH_VERBOSE=1

# This is the debhelper compatibility version to use.
export DH_COMPAT=1

build: build-stamp
build-stamp:
dh_testdir

# Add here commands to compile the package.
```

```
$(MAKE)

touch build-stamp

clean:
dh_testdir
dh_testroot
rm -f build-stamp

# Add here commands to clean up after the build process.
-$(MAKE) clean

dh_clean

install: build-stamp
dh_testdir
dh_testroot
dh_clean -k
dh_installdirs

# Add here commands to install the package into debian/tmp.
$(MAKE) install DESTDIR=`pwd`/debian/tmp

# Build architecture-independent files here.
binary-indep: build install
# We have nothing to do by default.

# Build architecture-dependent files here.
binary-arch: build install
# dh_testversion
dh_testdir
dh_testroot
# dh_installdebconf
dh_installdocs
dh_installexamples
dh_installmenu
# dh_installemacsen
# dh_installpam
# dh_installinit
dh_installcron
dh_installmanpages
dh_installinfo
# dh_undocumented
dh_installchangelogs
dh_link
dh_strip
```

```
63 dh_compress
64 dh_fixperms
65 # You may want to make some executables suid here.
66 dh_suidregister
67 # dh_makeshlibs
68 dh_installdeb
69 # dh_perl
70 dh_shlibdeps
71 dh_gencontrol
72 dh_md5sums
73 dh_builddeb
74
75 binary: binary-indep binary-arch
76 .PHONY: build clean binary-indep binary-arch binary install
```

(I've added the line numbers.)

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.

Lines 12 through 19 describe the 'build' (and its child 'build–stamp') rule, which runs the application's own Makefile to compile the program.

The 'clean' rule, as specified in lines 21–29, cleans up any unneeded binary or auto–generated stuff, left over from building the package. This rule must be working at all times (even when the source tree *is* cleaned up!), so please use the forcing options (e.g. for rm, that is '–f'), or ignore return values (with a '–' in front of a command name).

The installation process, the 'install' rule, starts with line 31. It basically runs the 'install' rule from the program's own Makefile, but installs in 'pwd'/debian/tmp directory – this is why we specified $(DESTDIR) as the root installation directory in gentoo's Makefile.

As the comments suggest, the 'binary–indep' rule, on the line 41, is used to build architecture independent packages. As we don't have any, nothing will be done there. If your package is 'Architecture: all' one, you need to include all the commands for building the package under this rule, and leave the next rule ('binary–arch') empty instead.

On to the next rule – 'binary–arch', on lines 45 through 73, in which we run several small utilities from the debhelper package that do various operations on your package files to make the package Policy conforming.

The names start with dh_, and the rest is the description of what the particular utility really does. It is all quite self–explanatory, but here are some additional explanations:

- `dh_testdir(1)` checks that you are in the right directory (i.e. the top–level source directory),
- `dh_testroot(1)` checks that you have root permissions which is needed for binary* targets, and clean,

- `dh_installmanpages(1)` copies all the manpages it can find in the source tree to package (beware, this is DWIM),
- `dh_strip(1)` strips debugging headers from executable files and libraries, to make them smaller,
- `dh_compress(1)` gzips man pages and documentation larger than 4 kB,
- `dh_installdeb(1)` copies package related files (e.g. the maintainer scripts) under debian/tmp/DEBIAN directory,
- `dh_shlibdeps(1)` calculates shared libraries dependencies of the libraries and executables,
- `dh_gencontrol(1)` adds stuff to, and installs, the control file,
- `dh_md5sums(1)` generates MD5 checksums for all the files in the package.

For more complete information on what do all these dh_* scripts do, and what are their other options, please read 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 binary–arch section is the one where you really should comment out any lines that call features you don't need. For gentoo, I'll comment out lines about testversion, emacsen, pam, init, cron, manpages, info, undocumented, suidregister, makeshlibs, and perl, simply because gentoo doesn't need them. Also, on the line 60, I'll need to add ' FIXES', because that is the name of the upstream changelog file.

The last two lines (along with any other not explained ones) are just some more–or–less necessary things, regarding which you can read in the make manual, and the Packaging Manual. For now, they're not important to know about.

Chapter 5

Other files under debian/

You will see that there exist several other files in the debian/ subdirectory, most of them with the '.ex' suffix, meaning that they are examples. If you wish or need to use any of those features, examine them and the related documentation (hint: the Policy Manual), rename them to remove the '.ex' suffix, and modify them and the rules file if necessary. Some of those files, those commonly used ones, are explained in the following sections.

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 <jrodin@jagor.srce.hr>, Wed, 11 Nov 1998 21:02:14 +0100
```

Since we don't have anything to put there – it is allowed to 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 will be prompted whether you want to keep your old configuration or not. You do this by entering the full path to each configuration file (they are usually in /etc,) one per line, in a file called conffiles.

Gentoo has one conffile, /etc/gentoorc, and we'll enter that in the 'conffiles' file. It is not actually necessary to have that file, if your program has no configuration files.

5.3 dirs

This file specifies the directories which we need but the normal installation procedure (make install) somehow doesn't create. By default, it looks like this:

```
usr/bin
usr/sbin
```

Note that the preceding slash is not included. We would have normally changed it to look like this:

```
usr/X11R6/bin
usr/X11R6/man/man1
```

but those directories are already created in the Makefile, so we won't need this file, and we may delete it.

5.4 manpage.1.ex

The files ending in *.ex are examples of how to add that kind of support into the package. To use one of them, edit it and remove the .ex extension. If you don't want to use it, just delete it.

Your program should have a man page. If it doesn't, this is a skeleton you can fill out. See the man pages for `man(7)` for a brief description of how to create a man page. Be sure to rename this file to the name of the program and make the extension the manual section it should go into. Here's a short list:

```
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 manpage should be called gentoo.1, or gentoo.1x because it is an X11 program. There was no gentoo.1 man page in the original source so I wrote it using information from the example and from upstream docs.

5.5 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. It isn't required by the Debian Policy, but users will surely appreciate it. We can add Gentoo to the menus by editing this file. Here's the default that dh_make creates:

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

The first field specifies what kind of interface the program needs (e.g. text or X11). The next is 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 third field is the name of the program. The fourth is the icon for the program or none if there isn't one. The fifth is the actual text which will appear in the menu. Finally, the sixth field is the command that runs the program.

Now we'll change the menu entry to this:

```
?package(gentoo):needs=X11 section=Apps/Misc \
  title="Gentoo" command="/usr/X11R6/bin/gentoo"
```

See `menufile(5)`, `update-menus(1)` and /usr/share/doc/debian–policy/menu–policy.html/ for more information.

5.6 watch.ex

You can use this file in addition to the `uscan(1)` and `uupdate(1)` programs (in the devscripts package) to watch the site you got the original source from. Here's what I put in it:

```
# watch control file for uscan
# Site Directory Pattern Version Script
ftp.obsession.se /gentoo gentoo-(.*)\.tar\.gz debian uupdate
```

Hint: connect to the Internet, and try running "uscan" in the program directory once you create the file. And read the manual pages.

5.7 ex.doc–base

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

This is how gentoo's doc–base file looks like:

```
Document: gentoo
Title: Gentoo Manual
Author: Emil Brink
Abstract: This manual describes what Gentoo is, and how it can be used.
Section: Apps/Tools

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/doc/doc–base/doc–base.html/index.html.

5.8 postinst.ex, preinst.ex, postrm.ex, prerm.ex

These files are called maintainer scripts, 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 Packaging Manual, section 6, and take a look at these example files provided by dh_make.

We should now be ready to build the package.

Chapter 6

Final steps

6.1 Building the package

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

```
dpkg-buildpackage -rfakeroot
```

This will do everything for you, you'll just have to enter your PGP secret key, twice. After it's done, you will see four new files in the directory above (~/debian/):

- *gentoo_0.9.12–1_i386.deb*

  This is the completed binary package. You can use dpkg or dselect to install and remove this just like any other package.

- *gentoo_0.9.12.orig.tar.gz*

  This is the original source code gathered up so that if someone else wants to recreate your package from scratch they can. Or if they aren't using the Debian packaging system, but need to manually download the source and compile.

- *gentoo_0.9.12–1.dsc*

  This is a summary of the contents of the source code. The file is generated from the gentoo–0.9.12/debian/control file, and is used when unpacking the source with `dpkg-source(1)`. This file is PGP 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)`.

- *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 gentoo–0.9.12/debian/changelog file and the .dsc file.

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. The long strings of numbers are MD5 checksums for the files mentioned. 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 hacked. This file is PGP signed, so that people can be even more sure that it's really yours.

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
```

Just make sure that your 'install' rule does **not** have 'install–stamp' child (that's the default nowadays), to make sure 'dh_clean –k' is run each time. And once you are finished with cleanups, remember to rebuild following the right procedure, to be able to upload correctly.

6.2 Checking the package for errors

Run `lintian(1)` on your .changes file; this program will check for many common packaging errors. The command is:

```
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 'Building the package' on the preceding page. If there are lines that begin with W:, those are only warnings, so you can be pretty much sure that your package is okay (but it most probably needs some tuning).

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

Look inside the package using a file manager like `mc(1)`, or unpack it in a temporary place using `dpkg-deb(1)`. Be on the lookout for extra unneeded files both in the binary and source package, in case something went wrong and some cruft didn't get cleaned up. 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' will list the files in the package.

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 on installation and running the program.

Later, when you build a new version, you should do the following to ensure basic upgradability of the package:

- upgrade from the previous version (and from the version in last Debian release),
- downgrade back again,
- install the package as a new package (i.e., with no previous version installed),
- uninstall it, reinstall it again, and then purge it.

6.3 Uploading the package

Now that you have tested your new package thoroughly, you'll need to upload these files to master.debian.org, using `dupload(1)`. First you have to set up dupload's config file, ~/.dupload.conf . Put something like this in it:

```
package config;
$default_host = "master";

$cfg{"master"}{"method"} = "scpb";
$cfg{"master"}{"login"} = "joy";
$cfg{"master"}{"visibleuser"} = "jrodin";
$cfg{"master"}{"visiblename"} = "jagor.srce.hr";
$cfg{"master"}{"fullname"} = "Josip Rodin";

$cfg{"non-us"}{"method"} = "scpb";
$cfg{"non-us"}{"login"} = "joy";
$cfg{"non-us"}{"visibleuser"} = "jrodin";
$cfg{"non-us"}{"visiblename"} = "jagor.srce.hr";
$cfg{"non-us"}{"fullname"} = "Josip Rodin";

1;
```

Of course, change my personal settings to yours, and read the `dupload.conf(5)` manual page to understand what each of these options means.

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

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

Dupload checks that the files' md5 checksums match those from the .changes file, so it will warn you to rebuild it as described in 'Building the package' on page 27 so it can properly upload.

Dupload will ask for your password on master.debian.org, upload the packages, and give a short announcement about your upload on <debian-devel-changes@lists.debian.org> if necessary.

If you live in Europe, you can use some other upload queues instead of master. For details look in `dupload(1)`, `dupload.conf(5)` and the Developer's Reference.

6.4 Updating the package

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 in the Debian changelog file, with 'dch –i', and include a short description of the bug and the solution, followed by this: "Closes: #54321". That way, the bug report will be automagically closed by the archive maintenance software the moment your package gets accepted in the Debian archive.
- Repeat what you did in 'Building the package' on page 27, 'Checking the package for errors' on page 28, and 'Uploading the package' on the preceding page. The difference is that this time, original source archive won't be included, as it hasn't been changed and it already exists in the Debian archive.

Now let's consider a different, a wee bit 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. ~/debian/).
- Enter the old source directory, and run:

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

  Of course, replace this filename with the name of your program's 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 'Building the package' on page 27, 'Checking the package for errors' on page 28, and 'Uploading the package' on the page before.

Note that if you set up 'debian/watch' file as described in 'watch.ex' on page 25, you can run `uscan(1)` to automagically look for revised sources, download them, and run uupdate.

6.5 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/package/* files and the man/info pages for all the programs mentioned in this article. When you receive a bug report (yes, actual bug reports!), you will know that it is time that you dig in the Debian Bug Tracking System (`http:`

//www.debian.org/Bugs/) and read the documentation there, to be able to deal with the reports efficiently.

By joining the Debian Mentors' mailing list at <debian-mentors@lists.debian.org> you can team up with experienced Debian developers who will help you with questions you might have. You can subscribe to it by sending e–mail to <debian-mentors-request@lists.debian.org> with the word 'subscribe' in the message subject.

If you still have questions, ask on the Debian Developers' mailing list at <debian-devel@lists.debian.org>. You can subscribe to it by sending e–mail to <debian-devel-request@lists.debian.org> with the word 'subscribe' in the message subject. If you are already a Debian developer, you should be subscribed to it anyway.

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 it will be fully in line with Debian policies (once again, read the *real documentation* for details). Good luck!