Debian policy manual

Ian Jackson <ijackson@gnu.ai.mit.edu>

version 2.1.1.0 (dpkg 1.4.0), 12 September 1996

0.1. Abstract

This manual describes the policy requirements which must be satisfied for a package to be included in the Debian distribution. This includes details of the permissions and ownerships of files in packages and other technical requirements as well as information like the upload procedure.

0.2. Contents

0.3. Copyright Notice

1. Introduction and scope of this manual

This manual describes the criteria that a Debian-format package must satisfy to be included in the Debian distribution.

Much of this information will be useful even when building a package which is to be distributed in some other way or is for local use.

This manual does *not* describe the technical mechanisms involved in package creation, installation and removal. This information can be found in the `dpkg` programmers' manual and the `dpkg` system administrators' manual.

This document assumes familiarity with these other two manuals. Unfortunately the system administrators' manual does not exist yet.

The Debian version of the FSF's GNU `hello` program is provided as an example for people wishing to create Debian packages.

Note that this document is still a draft!

2. Package copyright

Please study the copyright of your submission *carefully* and understand it before proceeding. If you have doubts or questions, please ask.

The aims of the policy detailed below are:

- That any user be able to rebuild any package in the official Debian distribution from the original source plus our patches.
- That we make available in our packaging formats as much software as we can.
- That it be easy for people to make CDROMs of our distribution without violating copyrights.

All packages in the Debian distribution proper must be freely useable, modifiable and redistributable in both source and binary form.[1] It must be possible for anyone to distribute and use modified source code and their own compiled binaries, at least when they do so as part of a Debian distribution.

Packages

- whose copyright permission notices (or patent problems) do not allow distribution and copying for profit, without restriction on the amount charged, or
- where distribution is restricted according to the medium used, or
- where the distributor must ask any kind of special permission of the authors, or
- with other onerous conditions,

may only be placed in the semi-supported non-free section of the Debian FTP archives. This is important so that CDROM manufacturers can distribute Debian without having to check the copyright of each package individually, simply by leaving out the contents of the non-free area; CDROM distributors are encouraged, though, to check the copyrights on programs in non-free individually and include as many as they can.

Packages

- whose copyright permission notices (or patent problems) allow only distribution of compiled binaries (and thus of which only binaries are available), or
- where the source code which may be distributed is not the complete source code required to compile the program (ie, the program cannot be compiled using only packages in the main Debian distribution), or

[1]It is OK for there to be a requirement that modified versions carry a warning, or that they be released with a different name or version number, or something similar, because we can comply with this requirement if necessary.

- which depend for their use on non-free or contrib packages[2], or
- allow free use only for a trial period (shareware), or
- are demonstration programs lacking vital functionality (crippleware), or
- are only installer-packages which require the user to supply a separate file to be installed, or
- which are buggy and no longer maintained, but are preserved for backward compatibility, or
- which fail to meet some other policy requirements,

may only be placed in the semi-supported contrib section of the Debian FTP archives (unless they need to be in non-free - see above).

Programs whose authors encourage the user to make donations are fine for the main distribution, provided that the authors do not claim that not donating is immoral, unethical, illegal or something similar; otherwise they must go in contrib (or non-free, if even distribution is restricted by such statements).

Packages whose copyright permission notices (or patent problems) do not allow redistribution even of only binaries, and where no special permission has been obtained, cannot placed on the Debian FTP site and its mirrors at all.

Note that under international copyright law[1] *no* distribution or modification of a work is allowed without an explicit notice saying so. Therefore a program without a copyright notice *is* copyrighted and you may not do anything to it without risking being sued! Likewise if a program has a copyright notice but no statement saying what is permitted then nothing is permitted.

Many authors are unaware of the problems that restrictive copyrights (or lack of copyright notices) can cause for the users of their supposedly-free software. It is often worthwhile contacting such authors diplomatically to ask them to modify their terms generally, or specially for Debian. However, this is a politically difficult thing to do and you should ask for advice on `debian-devel` first.

When in doubt, send mail to `<debian-devel@lists.debian.org>`. Be prepared to provide us with the copyright statement. Software covered by the GPL, public domain software and BSD-like copyrights are safe; be wary of the phrases ‘commercial use prohibited’ and ‘distribution restricted’.

Every package submission *must* be accompanied by verbatim copy of its copyright (with the exceptions of public domain packages and those covered by the UCB BSD licence or the GNU GPL or LGPL; in these cases simply indicate which is appropriate). This information must be included in a file installed by the binary package - see ‘`/usr/doc/`*package*`/copyright`’, page 10.

[2]This includes all packages which declare a dependency on a non-free or contrib package, and almost all packages which declare a recommendation.

[1]This applies in the United States, too.

3. Contents of the binary package

3.1. Control file requirements

3.1.1. `Maintainer` information

All packages must have a `Maintainer` field with the correct name and a working email address for the Debian maintainer of the package. If one person maintains several packages they should try to avoid having different forms of their name and address in different `Maintainer` fields.

3.1.2. Dependencies and virtual packages

Add a dependency for any shared libraries required by dynamically-linked executable binaries in your package, by using `dpkg-shlibdeps`.

All packages must use virtual package names where appropriate, and arrange to create new ones if necessary. They must not use virtual package names (except privately, amongst a cooperating group of packages) unless they have been agreed upon and appear in the list of virtual package names.

The latest version of the authoritative list of virtual package names can be found on `ftp.debian.org` in `/debian/doc/package-developer/virtual-package-names-list.text` or your local mirror. The procedure for updating it is described at the top of the file.

3.1.3. `Section` and `Priority`

Decide whether your package can go in `non-free`, `contrib` or the main distribution - see ‘Package copyright’, page 4, and put an appropriate value for the distribution in the `debian/changelog` file.

The `Priority` and `Section` control file fields give information for classifying the package in `dselect` and say which directory to place it in the FTP archive.

They are ultimately the responsibility of the distribution maintainers; however, you should suggest values for them in your `.changes` information when you upload a package. You do this by including appropriate information in the `debian/control` file before building the packages.

For a list of the currently in-use sections, please see the FTP archive. Packages in the non-free and contrib areas should have section `non-free` and `contrib`, respectively.

3.1.3.1. `Priority` values

`required`
: `required` packages are necessary for the proper functioning of the system. You must not

remove these packages or your system may become totally broken and you may probably not even be able to use `dpkg` to put things back. Systems with only the `required` packages are probably unuseable, but they do have enough functionality to allow the sysadmin to boot and install more software.

`important`
: Important programs, including those which one would expect to find on any Unix-like system. If the expectation is that an experienced Unix person who found it missing would go ‘What the F*!@<+ is going on, where is `foo`’, it should be in `important`. This is an important criterion because we are trying to produce, amongst other things, a free Unix. Other packages without which the system will not run well or be useable should also be here. This does *not* include Emacs or X11 or TeX or any other large applications. The `important` packages are just a bare minimum of commonly-expected and necessary tools.

`standard`
: These packages provide a reasonably small but not too limited character-mode system. This is what will install by default if the user doesn’t select anything else. It doesn’t include many large applications, but it does include Emacs (this is more of a piece of infrastructure than an application) and a reasonable subset of TeX and LaTeX (if this is possible without X).

`optional`[1]
: This is all the software that you might reasonably want to install if you didn’t know what it was or don’t have specialised requirements. This is a much larger system and includes X11, a full TeX distribution, and lots of applications.

`extra`
: This contains packages that conflict with others with higher priorities, or are only likely to be useful if you already know what they are or have specialised requirements.

Priority values are not case-sensitive.

3.1.3.2. Base packages

Some packages have `Section: base` and are in the `base` subdirectory on the FTP archives. These are the packages that are supplied on the base disks. They are the minimum sensible set for installing new packages (perhaps via a network).

Most of these packages should have `Priority: required` or at least `Priority: important`, and many will be essential (see below).

3.1.4. `Pre-Depends` and the `Essential` flag

Do not use `Pre-Depends` or `Essential: yes` unless your package is absolutely vital to the functioning of the system and the installation of other packages. Do not do either of these things without consultation with the distribution maintainers or on `debian-devel`.

Usually, neither of these fields should not be used unless removing a package really will completely hose the system, making it impossible to recover by (re)installing packages with

[1] In a sense everything is optional that isn’t required, but that’s not what is meant here.

dpkg.

Essential should not be used for a shared library package - the dependencies will prevent its premature removal, and we need to be able to remove it when it has been superseded.

It is not necessary for other packages to declare any dependencies they have on other packages which are marked Essential.

3.1.5. Including Priority and Section in the .deb control file

If a user installs a package which is not part of the standard distribution, or without downloading and updating from a new Packages file, the information about the priority and section of a package will be absent, and the dselect package listing will have the package listed under 'unclassified'. In order to improve this it is permissible to use the -is, -isp or -ip option to dpkg-gencontrol, so that the Section and/or Priority is copied into the actual control information in the .deb file. However, if you do this you should make sure you keep the information up to date so that users are not shown conflicting information.

3.1.6. Formatting of the Description control file field

Every Debian package should have an extended description.

The description should be written so that it tells the user what they need to know to decide whether to install the package. This description should not just be copied from the blurb for the program. Instructions for configuring or using the package should not be included - that is what installation scripts, manpages, Info files and /usr/doc/*package* are for. Copyright statements and other administrivia should not be included - that is what /usr/doc/*package*/copyright is for.

If you wish to include a list in your extended with entries which are a line or more each you must indent each entry by one space to make sure that it doesn't get wordwrapped. The start of each list entry should be marked with an asterisk, followed by a single space. You must wrap the list entries yourself to 75 columns, and should start continuation lines indented by three spaces so that they line up with the start of the text on the first line of each list entry.

See the programmers' manual for further requirements and pitfalls.

3.2. Locations of files

The location of all installed files and directories must comply fully with the Linux File System Standard (FSSTND). The latest version of this document can be found alongside this manual or on tsx-11.mit.edu in /pub/linux/docs/linux-standards/fsstnd/. Specific questions about following the standard may be asked on debian-devel, or referred to Daniel Quinlan, the FSSTND coordinator, at <quinlan@yggdrasil.com>.

3.2.1. Manpages

You must install manpages in nroff source form, in appropriate places under /usr/man. You should

only use sections 1 to 9 (see the FSSTND for more details). You must *not* install a preformatted 'cat page'.

If no manual page is available for a particular program, utility or function and this is reported as a bug on debian-bugs, a symbolic link from the requested manual page to the undocumented(7) manual page should be provided. This symbolic link can be created from debian/rules like this:

```
ln -s ../man7/undocumented.7 \
 debian/tmp/usr/man/man[1-9]/the_requested_manpage.[1-9]
```

This manpage claims that the lack of a manpage has been reported as a bug, so you may only do this if it really has (you can report it yourself, if you like). Do not close the bug report until a proper manpage is available.

You may forward a complaint about a missing manpage to the upstream authors, and mark the bug as forwarded in the Debian bug tracking system. Even though the GNU Project do not in general consider the lack of a manpage to be a bug, we do - if they tell you that they don't consider it a bug you should leave the bug in our bug tracking system open anyway.

Manpages should be installed compressed using gzip -9.

If one manpage needs to be accesssible via several names it is better to use a symbolic link than the .so feature, but there is no need to fiddle with the relevant parts of the upstream source to change from .so to symlinks - don't do it unless it's easy. Do not create hard links in the manual page directories, and do not put absolute filenames in .so directives. The filename in a .so in a manpage should be relative to the base of the manpage tree (usually /usr/man).

3.2.2. Info documents

Info documents should be installed in /usr/info. They should be compressed with gzip -9.

Your package must call install-info to update the Info dir file, in its post-installation script:

```
install-info --quiet --section Development Development \
  /usr/info/foobar.info
```

It is a good idea to specify a section for the location of your program; this is done with the --section switch. To determine which section to use, you should look at /usr/info/dir on your system and choose the most relevant (or create a new section if none of the current sections are relevant). Note that the --section flag takes two arguments; the first is a regular expression to match (case-insensitively) against an existing section, the second is used when creating a new one.

You must remove the entries in the pre-removal script:

```
install-info --quiet --remove /usr/info/foobar.info
```

If install-info cannot find a description entry in the Info file you will have to supply one. See install-info(8) for details.

3.2.3. Additional documentation

Any additional documentation that comes with the package can be installed at the discretion of the package maintainer. Text documentation should be installed in a directory `/usr/doc/`*package*[1] and compressed with `gzip -9` unless it is small.

If a package comes with large amounts of documentation which many users of the package will not require you should create a separate binary package to contain it, so that it does not take up disk space on the machines of users who do not need or want it installed.

It is often a good idea to put text information files (`README`s, changelogs, and so forth) that come with the source package in `/usr/doc/`*package* in the binary package. However, don't install the instructions for building and installing the package, of course!

3.2.4. Preferred documentation formats

The unification of Debian documentation is being carried out via HTML.

If your package comes with extensive documentation in a markup format that can be converted to various other formats you should if possible ship HTML versions in the binary package, in the directory `/usr/doc/`*package* or its subdirectories.

Other formats such as PostScript may be provided at your option.

3.2.4.1. Examples

Any examples (configurations, source files, whatever), should be installed in a directory `/usr/doc/`*package*`/examples`. These files should not be referenced by any program - they're there for the benefit of the system administrator and users, as documentation only.

3.2.5. `/usr/doc/`*package*`/changelog.Debian.gz` and `/usr/doc/`*package*`/changelog.upstream.gz`.

This installed file must contain a copy of the `debian/changelog` file from your Debian source tree, and a copy of the upstream changelog file if there is one. They should usually be installed in `/usr/doc/`*package* as `changelog.Debian.gz` and `changelog.gz` respectively.

Both should be installed compressed using `gzip -9`, as they will become large with time even if they start out small.

If the package has only one changelog which is used both as the Debian changelog and the upstream one because there is no separate upstream maintainer then that changelog should usually be installed as `/usr/doc/`*package*`/changelog.gz`; if there is a separate upstream maintainer, but no upstream changelog, then the Debian changelog should still be called `changelog.Debian.gz`.

3.2.6. `/usr/doc/`*package*`/copyright`

This file must contain details of the authorship and copyright of the package. It must say where the upstream sources (if any) were obtained, and explain briefly what modifications were made in the Debian version of the package compared to the upstream one. It must name the original

[1]Where *package* is the name of the package.

authors of the package and the Debian maintainer(s) who were involved with its creation.

It must contain the full text of the copyright notice and any acknowledgements for the program and the licence terms under which the program is distributed. If the package is distributed under the GNU General Public Licence, the GNU Library General Public Licence, the Regents of the University of California at Berkeley (BSD) licence or Larry Wall's Artistic Licence please say so instead of including a copy of the licence. The files `BSD`, `GPL`, `LGPL` and `Artistic` are be available in `/usr/doc/copyright` for you to refer to.

The copyright file should not be compressed unless it is very large.

Do not use the copyright file as a general `README` file. If your package has such a file it should be installed in `/usr/doc/`*package*`/README` or `README.Debian` or some other appropriate place.

3.2.7. Symbolic links

Most symbolic links should be relative, not absolute. Absolute links, in general, cause problems when a file system is not mounted where it "normally" resides (for example, when mounted via NFS).

In particular, symlinks from one part of `/usr` to another should be relative.

In certain cases, however, relative links may cause more problems. For example, links into `/etc` and `/var` should be absolute.

Note that when creating a relative link using `ln` it is not necessary for the target of the link to exist relative to the working directory you're running `ln` from; nor is it necessary to change directory to the directory where the link is to be made. Simply include the string that should appear as the target of the link (this will be a pathname relative to the directory in which the link resides) as the first argument to `ln`.

For example, in your `Makefile` or `debian/rules`, do things like:

```
ln -fs gcc $(prefix)/bin/cc
ln -fs gcc debian/tmp/usr/bin/cc
ln -fs ../sbin/sendmail $(prefix)/bin/runq
ln -fs ../sbin/sendmail debian/tmp/usr/bin/runq
```

3.2.8. Logfiles

Logfiles should usually be named `/var/log/`*package*`.log`. If you have many logfiles, or need a separate directory for permissions reasons (`/var/log` is writeable only by `root`), you should usually create a directory named `/var/log/`*package*.

Make sure that any logfiles are rotated occasionally so that they don't grow indefinitely; the best way to do this is to use `savelog` program in an `/etc/cron.daily`, `/etc/cron.weekly` or `/etc/cron.monthly` script.

Make sure that any logfiles are removed when the package is purged (but not when it is only removed), by checking the argument to the postrm script (see the programmer's manual for details).

3.2.9. `/usr/local` - for the use of the system administrator

As mandated by the FSSTND no package should place any files in `/usr/local`, either by putting them in the filesystem archive to be unpacked by `dpkg` or by manipulating them in their maintainer scripts.

Every package that searches a number of directories or files for something (for example, looking for shared libraries in `/lib` or `/usr/lib`) should search an appropriate directory in `/usr/local` too.

In order that the system administrator may know where to place additional files a package should create an empty directory in the appropriate place in `/usr/local` by supplying it in the filesystem archive for unpacking by `dpkg`. The `/usr/local` directory itself and all the subdirectories created by the package should have permissions 2775 (group-writeable and set-group-id) and be owned by `root.staff`.

In the future it will be possible to tell `dpkg` not to unpack files matching certain patterns, so that system administrators who do not wish these directories in `/usr/local` do not need to have them.

3.3. Permissions and ownerships

The rules in this section are guidelines for general use. If necessary you may deviate from the details below. However, if you do so you must make sure that what is done is secure and you must try to be as consistent as possible with the rest of the system. You should probably also discuss it on `debian-devel` first.

Files should be owned by `root.root`, and made writeable only by the owner and universally readable (and executable, if appropriate).

Directories should be mode 755 or (for group-writability) mode 2775. The ownership of the directory should be consistent with its mode - if a directory is mode 2775, it should be owned by the group that needs write access to it.

Setuid and setgid executables should be mode 4755 or 2755 respectively, and owned by the appropriate user or group. They should not be made unreadable (modes like 4711 or 2711 or even 4111); doing so achieves no extra security, because anyone can find the binary in the freely available Debian package - it is merely inconvenient. For the same reason you should not restrict read or execute permissions on non-set-id executables.

Some setuid programs need to be restricted to particular sets of users, using file permissions. In this case they should be owned by the uid to which they are set-id, and by the group which should be allowed to execute them. They should have mode 4754; there is no point in making them unreadable to those users who must not be allowed to execute them.

Do not arrange that the system administrator can only reconfigure the package to correspond to their local security policy by changing the permissions on a binary. Ordinary files installed by `dpkg` (as opposed to conffiles and other similar objects) have their permissions reset to the distributed

permissions when the package is reinstalled. Instead you should consider (for example) creating a group for people allowed to use the program(s) and making any setuid executables executable only by that group.

Shared libraries should be installed executable.

3.4. Configuration files

Any configuration files created or used by your package should reside in `/etc`. If there are several you should consider creating a subdirectory named after your package.

It is almost certain that any file in `/etc` that is in your package's filesystem archive should be listed in `dpkg`'s `conffiles` control area file. (See the `dpkg` programmers' manual).

3.5. Maintainer scripts

The package installation scripts should avoid producing output which it is unnecessary for the user to see and should rely on `dpkg` to stave off boredom on the part of a user installing many packages. This means, amongst other things, using the `--quiet` option on `install-info`.

Packages should try to minimise the amount of prompting they need to do, and they should ensure that the user will only every be asked each question once. This means that packages should try to use appropriate shared configuration files (such as `/etc/papersize` and `/etc/news/server`, rather than each prompting for their own list of required pieces of information.

It also means that an upgrade should not ask the same questions again, unless the user has used `dpkg --purge` to remove the package's configuration. The answers to configuration questions should be stored in an appropriate place in `/etc` so that the user can modify them, and how this has been done should be documented.

If a package has a vitally important piece of information to pass to the user (such as "don't run me as I am, you must edit the following configuration files first or you risk your system emitting badly-formatted messages"), it should display this in the `postinst` script and prompt the user to hit return to acknowledge the message. Copyright messages do not count as vitally important (they belong in `/usr/doc/copyright`); neither do instructions on how to use a program (these should be in on line documentation, where all the users can see them).

Any necessary prompting should almost always be confined to the post-installation script, and should be protected with a conditional so that unnecssary prompting doesn't happen if a package's installation fails and the `postinst` is called with `abort-upgrade`, `abort-remove` or `abort-deconfigure`.

Errors which occur during the execution of an installation script *must* be checked and the installation *must not* continue after an error.

The section below on scripts in general applies to package maintainer scripts too.

3.6. Scripts in general

All command scripts, including the package maintainer scripts inside the package and used by `dpkg`, should have a `#!` line naming the shell to be used to interpret them.

In the case of Perl scripts this should be `#!/usr/bin/perl`.

Shell scripts (`sh` and `bash`) should almost certainly start with `set -e` so that errors are detected. Every script *must* use `set -e` or check the exit status of *every* command.

Perl scripts should check for errors when making any system calls, including `open`, `print`, `close`, `rename` and `system`.

`csh` and `tcsh` should be avoided as scripting languages. See Csh Programming Considered Harmful, one of the `comp.unix.*` FAQs. If an upstream package comes with `csh` scripts then you must make sure that they start with `#!/bin/csh` and make your package depend on `csh`.

3.7. Shared library packages

Packages involving shared libraries should be split up into several binary packages.

For a straightforward library which has a development environment and a runtime kit including just shared libraries you need to create two packages: *librarynamesoname*[1] and *librarynamesoname*`-dev`.

If you prefer only to support one development version at a time you may name the development package *libraryname*`-dev`; otherwise you may wish to use `dpkg`'s conflicts mechanism to ensure that the user only installs one development version at a time (after all, different development versions are likely to have the same header files in them, causing a filename clash if both are installed). Typically the development version will also need an exact version dependency on the runtime library, to make sure that compilation and linking happens correctly.

Packages which use the shared library should have a dependency on the name of the shared library package, *librarynamesoname*. When the *soname* changes you can have both versions of the library installed while moving from the old library to the new.

If your package has some run-time support programs which use the shared library you must *not* put them in the shared library package. If you do that then you won't be able to install several versions of the shared library without getting filename clashes. Instead, either create a third package for the runtime binaries (this package might typically be named *libraryname*`-runtime` - note the absence of the *soname* in the package name) or if the development package is small include them in there.

If you have several shared libraries built from the same source tree you can lump them all togther into a single shared library package, provided that you change all their *soname*s at once (so that

[1] *soname* is the shared object name of the shared library - it's the thing that has to match exactly between building an executable and running it for the dynamic linker to be able run the program. Usually the *soname* is the major number of the library.

you don't get filename clashes if you try to install different versions of the combined shared libraries package).

Follow the directions in the `dpkg` programmers' manual for putting the shared library in its package, and make sure you include a `shlibs` control area file with details of the dependencies for packages which use the library.

3.8. Application configuration files, dotfiles and `/etc/skel`

Files in `/etc/skel` will automatically be copied into new user accounts by `adduser`. They should not be referenced there by any program.

Therefore, if a program needs a dotfile to exist in advance in `$HOME` to work sensibly that dotfile should be installed in `/etc/skel` (and listed in conffiles, if it is not generated and modified dynamically by the package's installation scripts).

However, programs that require dotfiles in order to operate sensibly (dotfiles that they do not create themselves automatically, that is) are a bad thing, and programs should be configured by the Debian default installation as close to normal as possible.

Therefore, if a program in a Debian package needs to be configured in some way in order to operate sensibly that configuration should be done in a site-wide global configuration file elsewhere in `/etc`. Only if the program doesn't support a site-wide default configuration and the package maintainer doesn't have time to add it should a default per-user file be placed in `/etc/skel`.

`/etc/skel` should be as empty as we can make it. This is particularly true because there is no easy mechanism for ensuring that the appropriate dotfiles are copied into the accounts of existing users when a package is installed.

Ideally the sysadmin should ideally not have to do any configuration other than that done (semi-)automatically by the postinst script.

3.9. Games

The permissions on /var/lib/games are 755 `root.root`.

Each game decides on its own security policy.

Games which require protected, privileged access to high-score files, savegames, &c, must be made set-*group*-id (mode 2755) and owned by `root.games`, and use files and directories with appropriate permissions (770 `root.games`, for example). They must *not* be made set-*user*-id, as this causes security problems.[1]

Some packages, for example some fortune cookie programs, are configured by the upstream authors to install with their data files or other static information made unreadable so that they

[1] If an attacker can subvert any set-user-id game they can overwrite the executable of any other, causing other players of these cames to run a trojan. With a set-group-id game the attacker only gets access to less important game data, and if they can get at the other players' accounts at all it will take considerably more effort.

can only be accessed through set-id programs provided. Do not do this in a Debian package: anyone can download the `.deb` file and read the data from it, so there is no point making the files unreadable. Not making the files unreadable also means that you don't have to make so many programs set-id, which reduces the risk of a security hole.

3.10. Allocating package-specific users and groups

If you need to create a new user or group for your package there are two possibilities. Firstly, you may need to make some files in the binary package be owned by this user or group, or you may need to compile the user or group id (rather than just the name) into the binary (though this latter should be avoided if possible). In this case you need a statically allocated id.

You must ask for a user or group id from the base system maintainer, and must not release the package until you have been allocated one. Once you have been allocated one you must make the package depend on a version of the base system with the id present in `/etc/passwd` or `/etc/group`, or alternatively arrange for your package to create the user or group itself with the correct id (using `adduser`) in its pre- or post-installation script (the latter is to be preferred if it is possible).

On the other hand, the program may able to determine the uid or gid from the group name at runtime, so that a dynamic id can be used. In this case you must choose an appropriate user or group name, discussing this on `debian-devel` and checking with the base system maintainer that it is unique and that they do not wish you to use a statically allocated id instead. When this has been checked you must arrange for your package to create the user or group if necessary using `adduser` in the pre- or post-installation script (again, the latter is to be preferred if it is possible).

Note that changing the numeric value of an id associated with a name is very difficult, and involves searching the filesystem for all appropriate files. You need to think carefully whether a static or dynamic id is required, since changing your mind later will cause problems.

3.11. Installation of Emacs-lisp files

Generally, if a package includes an elisp helper file, it probably doesn't need to be byte-compiled. If the package is *written* primarily in emacs, it is probably complex enough that speed is an issue and should be byte compiled.

3.12. Use of `dpkg-divert` and `update-alternatives`

Do not use `dpkg-divert` on a file belonging to another package without consulting the maintainer of that package first.

In order for `update-alternatives` to work correctly all the packages which supply an instance of the 'shared' command name (or, in general, filename) must use it. You can use `Conflicts` to force the deinstallation of other packages supplying it which do not (yet) use `update-alternatives`. It may in this case be appropriate to specify a conflict on earlier versions on something - this is an exception to the usual rule that this is not allowed.

4. Configuring and building the programs in the package

4.1. Compilation options

Generally the following compilation parameters should be used:

```
CC = gcc
CFLAGS = -O2 -g -Wall # sane warning options vary between programs
LDFLAGS = # none
install -s # (or use strip on the files in debian/tmp)
```

Note that all installed binaries and shared libraries should be stripped[1,2], either by using the `-s` flag to `install`, or by calling `strip` on the binaries after they have been copied into `debian/tmp` but before the tree is made into a package.

Make sure that you do not link with `-g`, as this makes a.out compilers produce huge statically linked binaries. The `-g` flag is useful on compilation so that you have available a full set of debugging symbols in your built source tree, in case anyone should file a bug report involving (for example) a core dump.

The `-N` flag should not be used. On a.out systems it may have been useful for some very small binaries, but for ELF it has no good effect.

It is up to the package maintainer to decide what compilation options are best for the package.

[1] Shared libraries can function perfectly well when stripped, since the symbols for dynamic linking are in a separate part of the ELF object file.

[2] Under some circumstances it may be useful to install a shared library unstripped, for example when building a separate package to support debugging.

Certain binaries (such as computationally-intensive programs) may function better with certain flags (`-O3`, for example); feel free to use them. Please use good judgment here. Don't use flags for the sake of it; only use them if there is good reason to do so. Feel free to override the upstream author's ideas about which compilation options are best - they are often inappropriate for our environment.

Please make sure that you use only released versions of shared libraries to build your packages; otherwise other users will not be able to run your binaries properly. Producing source packages that depend on unreleased compilers is also usually a bad idea.

4.2. Packages which can use the X shared libraries

Some programs can be configured with or without support for X Windows. Typically these binaries produced when configured for X will need the X shared libraries to run.

Such programs should be configured *with* X support, and should declare a dependency on **elf-x11r6lib** (for the X11R6 libraries). Users who wish to use the program can install just the relatively small **xlib** package, and do not need to install the whole of X.

Do not create two versions (one with X support and one without) of your package.

4.3. Mail processing on Debian systems

Debian packages which process electronic mail, whether mail-user-agents (MUAs) or mail-transport-agents (MTAs), *must* make sure that they are compatible with the configuration decisions below. Failure to do this may result in lost mail, broken `From:` lines, and other serious brain damage!

The mail spool is `/var/spool/mail` and the interface to send a mail message is `/usr/sbin/sendmail` (as per the FSSTND). The mail spool is part of the base system and not part of the MTA package.

Mailboxes are locked using the *username*`.lock` lockfile convention, rather than `fcntl`, `flock` or `lockf`.

Mailboxes are generally 660 *user*`.mail` unless the user has chosen otherwise. A MUA may remove a mailbox (unless it has nonstandard permissions) in which case the MTA or another MUA must recreate it if needed. Mailboxes must be writeable by group mail.

The mail spool is 2775 `mail.mail`, and MUA's need to be setgid mail to do the locking mentioned above (and obviously need to avoid accessing other users' mailboxes using this privilege).

`/etc/aliases` is the source file for the system mail aliases (e.g. postmaster, usenet, etc.) - it is the one which the sysadmin and postinst scripts may edit. After `/etc/aliases` is edited the program or human editing it must call `newaliases`. All MTA packages should come with a `newaliases` program, even if it does nothing, but older MTA packages do not do this so programs should not fail if `newaliases` cannot be found.

The convention of writing `forward to` *address* in the mailbox itself is not supported. Use a `.forward` file instead.

The location for the `rmail` program used by UUCP for incoming mail is `/usr/sbin/rmail`, as per the FSSTND. Likewise, `rsmtp`, for receiving batch-SMTP-over-UUCP, is in `/usr/sbin/rsmtp` if it is supported.

If you need to know what name to use (for example) on outgoing news and mail messages which are generated locally, you should use the file `/etc/mailname`. It will contain the portion after the username and `@` (at) sign for email addresses of users on the machine (followed by a newline).

A package should check for the existence of this file. If it exists it should use it without comment.[1] If it does not exist it should prompt the user for the value and store it in `/etc/mailname` as well as using it in the package's configuration. The prompt should make it clear that the name will not just be used by that package. E.g., in this situation the INN package says:

```
Please enter the 'mail name' of your system.  This is the hostname
portion of the address to be shown on outgoing news and mail messages.
The default is syshostname, your system's host name.
Mail name ['syshostname']:
```

where *syshostname* is the output of `hostname -fqdn`.

4.4. Obsolete constructs and libraries: varargs and libtermcap

`<varargs.h>` is provided to support end-users compiling very old software; libtermcap is provided to support the execution of software which has been linked against it (either old programs or those such as Netscape which are only available in binary form).

Debian packages should be ported to `<stdarg.h>` and ncurses when they are built.

[1] An MTA's prompting configuration script may wish to prompt the user even if it finds this file exists.

5. Source packaging, changelogs, etc.

5.1. Releases of packages by other than the usual Debian maintainer

Under certain circumstances it is necessary for someone other than the usual package maintainer to make a release of a package. For example, a porter for another architecture may have to make some small changes to the source package and does not wish to wait with uploading their release until the main maintainer has incorporated the patch, or a serious security problem may have come to light requiring immediate attention.

Maintainers other than the usual package maintainer should make as few changes to the package as possible, and they should always send a unified context diff (`diff -u`) detailing their changes to the bug tracking system properly flagged with the correct package so that the usual maintainer is kept aware of the situation.

When someone other than the usual maintainer releases a package they should add a new component to the *debian-revision* component of the version number - that is, the portion after the (last) hyphen. This extra component will start at `1`. This is to avoid 'stealing' one of the usual maintainer's version numbers, possibly disrupting their work. If there is no *debian-revision* component in the version number then one should be created, starting at `1`.

If it is absolutely necessary for someone other than the usual maintainer to make a release based on a new upstream version then the person making the release should start with the *debian-revision* value `0.1`. The usual maintainer of a package should start their *debian-revision* numbering at `1`.

5.2. Standards conformance and `Standards-Version`

You should specify the most recent version of the packaging standards with which your package complies in the source package's `Standards-Version` field.

This value will be used to file bug reports automatically if your package becomes too much out of date.

The value corresponds to a version of the Debian manuals, as can be found on the title page or page headers and footers (depending on the format). The value for this version of the manuals and packaging standards is `2.1.1.0`.

The version number has four components - major and minor number and major and minor patchlevel. When the standards change in a way that requires every package to change the major number will be changed. Significant changes that will require work in many packages will be signaled by a change to the minor number. The major patchlevel will be changed for any change to the meaning of the standards, however small; the minor patchlevel will be changed when only cosmetic, typographical or other edits which do not change the meaning are made, or changes which do not affect the contents of packages.

You should regularly, and especially if your package has become out of date, install the most recent version of dpkg and read `/usr/doc/dpkg/changelog-manuals` to see which changes, if any, are relevant. If any are relevant you should look up the relevant section in the policy or programmers' manuals and update your package. When your package complies with the new standards you may update the `Standards-Version` source package field and release it.

5.3. Documentation and the `changelog`

Document your changes and updates to the source package properly in the `debian/changelog` file.

A copy of the file which will be installed in `/usr/doc/`*package*`/copyright` should be in debian/copyright.

In non-experimental packages you may only use a format for debian/changelog which is supported by the most recent released version of dpkg. If your format is not supported and there is general support for it you should contact the dpkg maintainer to have the parser script for your format included in the dpkg package.[1]

5.4. Changes to the upstream sources

If you need to edit a Makefile where GNU-style configure scripts are used, you should edit the .in files rather than editing the Makefile directly. This allows the user to reconfigure the package if necessary. You should *not* configure the package and edit the generated Makefile! This makes it impossible for someone else to later reconfigure the package.

If changes to the source code are made that are generally applicable please try to get them included in the upstream version of the package by supplying the upstream authors with the changes in whatever form they prefer.

If you need to configure the package differently for Debian or for Linux, and the upstream source doesn't provide a way to configure it the way you need to, please add such configuration facilities (for example, a new autoconf test or #define) and send the patch to the upstream authors, with the default set to the way they originally had it. You can then easily override the default in your debian/rules or wherever is appropriate.

5.5. Error trapping in makefiles

When make invokes a command in a makefile (including your package's upstream makefiles and the debian/rules) it does so using sh. This means that sh's usual bad error handling properties apply: if you include a miniature script as one of the commands in your makefile you'll find that if you don't do anything about it then errors are not detected and make will blithely continue after problems.

Every time you put more than one shell command (this includes using a loop) in a makefile command you *must* make sure that errors are trapped. For simple compound commands, such as changing directory and then running a program, using && rather than semicolon as a command separator is sufficient. For more complex commands including most loops and conditionals you must include a separate set -e command at the start of every makefile command that's actually one of these miniature shellscripts.

[1] You will need to agree that the parser and its manpage may be distributed under the GNU GPL, just as the rest of dpkg is.

6. Procedure - how to participate in the Debian project

6.1. Before you start work on a package

So, you've read all the documentation, you understand what everything in the hello example package is for, and you're about to Debianise your favourite package. How do you actually become a Debian developer so that your work can be incorporated into the Project?

Firstly, subscribe to debian-devel if you haven't already. Send the word subscribe in the *Subject* of a mail to <debian-devel-REQUEST@lists.debian.org>. In case of problems contact the list administrator, Anders Chrigstrom <ac@netg.se>.

You should to subscribe and lurk for a bit before doing any coding, and you should post about your intentions to work on something to avoid duplicated effort.

If you do not have a PGP key yet generate one. You should probably read the PGP manual, as it has much important information which is critical to its security. Many more security failures are due to human error than to software failure or high-powered spy techniques.

If you live in a country where use of cryptography even for signing is forbidden then please contact us so we can make special arrangements. This does not apply in France, where I believe only encryption and not signing is forbidden.

6.2. When you have a package to upload

When you have your package ready to be uploaded you must send a message to the project leader, Bruce Perens **<bruce@pixar.com>**, the administrator of **master.debian.org**, Simon Shapiro **<shimon@i-connect.net>**, the mailing list administrator, Anders Chrigstrom **<ac@netg.se>** and the **dpkg** maintainer, Ian Jackson **<ijackson@gnu.ai.mit.edu>**.

The message should say what you've done and who you are, and should ask for an account on **master** and to be subscribed to **debian-private** (the developers-only mailing list). It should contain your PGP key (extracted using **pgp -kxa**) for the database of keys which is shipped with **dpkg**. When you have your personal account on **master** log in and transfer the files to **/home/Debian/ftp/private/project/Incoming**. You cannot upload to **Incoming** on **master** using anonymous FTP.

You can also upload files to **Incoming** via a **cron**-driven upload queue in Europe on **chiark.chu.cam.ac.uk**. For details connect to **chiark** using anonymous FTP and read **/pub/debian/private/project/README.how-to-upload**.

6.3. Upload handling - `.changes` files

When a package is uploaded to the Debian FTP archive, it must be accompanied by a **.changes** file which gives directions for its handling. This is usually generated by **dpkg-genchanges**.

This file is a control file with the following fields:
- **Format**
- **Date**
- **Source**
- **Binary**
- **Architecture**
- **Version**
- **Distribution**
- **Urgency**
- **Maintainer**
- **Description**
- **Changes**

- **Files**

All of them are mandatory for a Debian upload. See the list of control fields in the **dpkg** programmers' manual for the contents of these fields.

The first time a version is uploaded which corresponds to a particular upstream version the original source tarfile should be uploaded and included in the **.changes** file; subsequent times the very same tarfile should be used to build the new diffs and **.dsc** files, and it need not then be uploaded.

By default **dpkg-genchanges** and **dpkg-buildpackage** will include the original source tarfile if and only if the Debian revision part of the source version number is **0** or **1**, indicating a new upstream version. This behaviour may be modified by using **-sa** to always include it or **-sd** to always leave it out.

If no original source is included in the upload then the original source tarfile used by **dpkg-source** when constructing the **.dsc** file and diff to be uploaded *must* be byte-for-byte identical with the one already in the archive. If there is some reason why this is not the case then the new version of the original source should be uploaded, possibly by using the **-sa** flag.

6.4. Upload handling - announcements

When a package is uploaded an announcement should be posted to **debian-changes**. The announcement should give the (source) package name and version number, and a very short summary of the changes, in the **Subject** field, and should contain the PGP-signed **.changes** file. Some additional explanatory text may be added before the start of the **.changes** file.

If a package is released with **Distribution: experimental** the announcement should be posted to **debian-devel** instead.

6.5. The Debian mailing lists

The mailing list server is at **lists.debian.org**. Mail **debian-***foo***-REQUEST@lists.debian.org**[1] with the word **subscribe** in the Subject to subscribe or **unsubscribe** to unsubscribe.

When replying to messages on the mailing list, please do not send a carbon copy (**CC** - this does not mean 'courtesy copy') to the original poster. Anyone who posts to a mailing list should read it to see the responses.

As ever on the net, please trim down the quoting of articles you're replying to. In general, please adhere to the usual conventions for posting messages.

[1] where **debian-***foo* is the name of the list

7. Conversion procedure from old source packages

This is a brief summary of the procedure for converting a pre-2.0.0.0-format source package into the new format.

You are strongly advised to download and examine the `hello` package, and to read the section in the `dpkg` programmers' manual describing the source packaging tools. More detail about the exact functionality of these tools is available in `dpkg-source(1)`.

- Download the original source code from wherever it can be found and do any rearrangement required to make it look like the original tree of the Debian source. Put it in *package-upstream-version*`.orig/` or *package_upstream-version*`.orig.tar.gz`.
- Rename all files `debian.*` to `debian/*`. There may be some exceptions to this, but this is a good start.
- Edit the `debian/changelog` - create or rename it if necessary. Add a new revision to the top with the appropriate details, and a local variables entry to the bottom to set Emacs to the right mode:

  ```
  Local variables:
  mode: debian-changelog
  End:
  ```
- Edit/create `debian/control`:
  - Remove the `Version` field. If it is generated unusually (not equal to the source version) you must use the -v option to dpkg-gencontrol (see below). `Section`, `Priority`, `Maintainer` go above the first blank line, most of the rest below.
  - Reorder the fields and add a blank line at an appropriate point, separating the source package fields from the binary package fields.
  - Add the `Source` field.
  - Add the `Standards-Version` field. The current value is `2.1.1.0`.
  - Change the `Architecture` field for each package to `any`, `all` or whatever. If there isn't an `Architecture` field add one.
  - If any other seddery or things used to happen to make the binary control files use `dpkg-gencontrol`'s variable substitution features to achieve the same effect. Use `debian/substvars` if you need to put unusally-generated information (apart from details of `.deb` files) in the `.changes` file too.
- Edit the `debian/rules`:
  - Remove the `source` and `diff` and any `changes` and `dist` targets. These things now happen in a package-independent way and are not done by `debian/rules`.
  - Split the `binary` target into `binary-arch` and `binary-indep`; in many cases all of `binary` should go into `binary-arch`. Create the `binary` target and the unused of the two other `binary-*` targets if there is one - you can copy the ones from the `hello` package.
  - Change the `binary` target to use `dpkg-gencontrol` to make the package control file(s). Move it to after all the files have been installed but just before the last `chown` and `chmod` in the target.
  - Change occurrences of `debian-tmp` to `debian/tmp`.
  - Change occurrences of `debian.{post,pre}{inst,rm}` to `debian/*`.
  - Remove the version number setting at the top, if there is one.
  - Ensure that the package's Debian-specific and upstream changelogs are installed.

- Change the package to use `dpkg-shlibdeps` to determine its shared library dependencies and substitute them in. Shared library dependencies should no longer be hardwired in the source package.
- Check that the `debian/README` is really the copyright file, and if so rename it to `debian/copyright` and edit `debian/rules` to cope with this and to change the installation of the copyright file from `/usr/doc/`*package*`/copyright` instead of `/usr/doc/copyright/`*package*. If it isn't then find `debian/copyright` and decide what to do with the `README`.
- Check for various other anachronisms and problems:
  - Remove any `Package_Revision`, `Package-Revision` or `Revision` fields.
  - Rename `Optional` to `Suggests`, `Recommended` to `Recommends`.
  - Change `/usr/doc/examples/`*package* to `/usr/doc/`*package*`/examples`.
  - Make sure that manpages are installed compressed.
  - Check that the description has an extended description, is well-formatted and meaningful and helpful to people wanting to know whether to install a package.
- Look everything over.
- Do a test build using `dpkg-buildpackage -ur -uc -sa -r`*whatever*. Check the permissions and locations of files in the resulting package by eyeballing the output of `dpkg-deb --contents`, and check that the source build happened OK. Test install the binary package(s) and test extract the source package(s).
- Sign the release: either rebuild everything with `dpkg-buildpackage -sa`, or PGP-sign the `.dsc`, rebuild the `.changes` using `dpkg-genchanges -sa`, and then PGP-sign the `.changes`.

The use of `-sa` on `dpkg-buildpackage` and `dpkg-genchanges` is important when doing the first build/uploading of a new-format source package. Unless this happens to be Debian revision `0` or `1` by default the original source tarfile will not be included in the uploaded files listed in the `.changes` file, and so it won't be installed on the FTP site. `-sa` requests that the original source be included regardless.