Debian Policy Manual

Ian Jackson <ijackson@gnu.ai.mit.edu>
Christian Schwarz <schwarz@debian.org>
revised: David A. Morris <bweaver@debian.org>
The Debian Policy mailing List <debian-policy@lists.debian.org>

version 3.5.6.0, 2001-07-24

Abstract

This manual describes the policy requirements for the Debian GNU/Linux distribution. This includes the structure and contents of the Debian archive and several design issues of the operating system, as well as technical requirements that each package must satisfy to be included in the distribution. The policy package itself is maintained by a group of maintainers that have no editorial powers. The current list of maintainers is:

1. Julian Gilbey <jdg@debian.org>
2. Manoj Srivastava <srivasta@debian.org>

Copyright Notice

Contents

Chapter 1

About this manual

1.1 Scope

This manual describes the policy requirements for the Debian GNU/Linux distribution. This includes the structure and contents of the Debian archive and several design issues of the operating system, as well as technical requirements that each package must satisfy to be included in the distribution.

This manual also describes Debian policy as it relates to creating Debian packages. It is not a tutorial on how to build packages, nor is it exhaustive where it comes to describing the behavior of the packaging system. Instead, this manual attempts to define the interface to the package management system that the developers have to be conversant with.[1]

The footnotes present in this manual are merely informative, and are not part of Debian policy itself.

In this manual, the words *must*, *should* and *may*, and the adjectives *required*, *recommended* and *optional*, are used to distinguish the significance of the various guidelines in this policy document. Packages that do not conform to the guidelines denoted by *must* (or *required*) will generally not be considered acceptable for the Debian distribution. Non-conformance with guidelines denoted by *should* (or *recommended*) will generally be considered a bug, but will not necessarily render a package unsuitable for distribution. Guidelines denoted by *may* (or *optional*) are truly optional and adherence is left to the maintainer's discretion.

These classifications are roughly equivalent to the bug severities *serious* (for *must* or *required* directive violations), *minor*, *normal* or *important* (for *should* or *recommended* directive violations) and *wishlist* (for

[1] Informally, the criteria used for inclusion is that the material meet one of the following requirements:

Standard interfaces The material presented represents an interface to the packaging system that is mandated for use, and is used by, a significant number of packages, and therefore should not be changed without peer review. Package maintainers can then rely on this interfaces not changing, and the package management software authors need to ensure compatibility with these interface definitions. (Control file and changelog file formats are examples.)

Chosen Convention If there are a number of technically viable choices that can be made, but one needs to select one of these options for inter-operability. The version number format is one example.

Please note that these are not mutually exclusive; selected conventions often become parts of standard interfaces.

optional items).[2]

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

1.2 New versions of this document

The current version of this document is always accessible from the Debian FTP server `ftp.debian.org` as `/debian/doc/package-developer/policy.txt.gz` (also available from the same directory are several other formats: `policy.html.tar.gz`, `policy.pdf.gz` and `policy.ps.gz`) or from the Debian Policy Manual (`http://www.debian.org/doc/debian-policy/`) webpage.

In addition, this manual is distributed via the Debian package `debian-policy`.

The `debian-policy` package also includes the file `upgrading-checklist.txt` which indicates policy changes between versions of this document.

1.3 Feedback

As the Debian GNU/Linux system is continuously evolving this manual does so too.

While the authors of this document have tried hard to avoid typos and other errors, these do still occur. If you discover an error in this manual or if you want to give any comments, suggestions, or criticisms please send an email to the Debian Policy List, `<debian-policy@lists.debian.org>`, or submit a bug report against the `debian-policy` package.

[2] Compare RFC 2119. Note, however, that these words are used in a different way in this document.

Chapter 2

The Debian Archive

The Debian GNU/Linux system is maintained and distributed as a collection of *packages*. Since there are so many of them (currently well over 6000), they are split into *sections* and given *priorities* to simplify the handling of them.

The effort of the Debian project is to build a free operating system, but not every package we want to make accessible is *free* in our sense (see the Debian Free Software Guidelines, below), or may be imported/exported without restrictions. Thus, the archive is split into the sections *main*, *non-free*, *contrib*, *non-US/main*, *non-US/non-free*, and *non-US/contrib*. The sections are explained in detail below.

The *main* and the *non-US/main* sections together form the *Debian GNU/Linux distribution*.

Packages in the other sections are not considered to be part of the Debian distribution, although we support their use and provide infrastructure for them (such as our bug-tracking system and mailing lists). This Debian Policy Manual applies to these packages as well.

2.1 Package copyright and sections

The aims of this section are:

- to allow us to make as much software available as we can,
- to allow us to encourage everyone to write free software, and
- to allow us to make it easy for people to produce CD-ROMs of our system without violating any licenses, import/export restrictions, or any other laws.

2.1.1 The Debian Free Software Guidelines

The Debian Free Software Guidelines (DFSG) form our definition of ‘free software’. These are:

Free Redistribution The license of a Debian component may not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license may not require a royalty or other fee for such sale.

Source Code The program must include source code, and must allow distribution in source code as well as compiled form.

Derived Works The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.

Integrity of The Author's Source Code The license may restrict source-code from being distributed in modified form *only* if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software. (This is a compromise. The Debian Project encourages all authors to not restrict any files, source or binary, from being modified.)

No Discrimination Against Persons or Groups The license must not discriminate against any person or group of persons.

No Discrimination Against Fields of Endeavor The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.

Distribution of License The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.

License Must Not Be Specific to Debian The rights attached to the program must not depend on the program's being part of a Debian system. If the program is extracted from Debian and used or distributed without Debian but otherwise within the terms of the program's license, all parties to whom the program is redistributed must have the same rights as those that are granted in conjunction with the Debian system.

License Must Not Contaminate Other Software The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be free software.

Example Licenses The "GPL," "BSD," and "Artistic" licenses are examples of licenses that we consider *free*.

2.1.2 The main section

Every package in *main* and *non-US/main* must comply with the DFSG (Debian Free Software Guidelines).

In addition, the packages in *main*

- must not require a package outside of *main* for compilation or execution (thus, the package must not declare a “Depends”, “Recommends”, or “Build-Depends” relationship on a non-*main* package),
- must not be so buggy that we refuse to support them, and
- must meet all policy requirements presented in this manual.

Similarly, the packages in *non-US/main*
- must not require a package outside of *main* or *non-US/main* for compilation or execution,
- must not be so buggy that we refuse to support them,
- must meet all policy requirements presented in this manual.

2.1.3 The contrib section

Every package in *contrib* and *non-US/contrib* must comply with the DFSG.

In addition, the packages in *contrib* and *non-US/contrib*
- must not be so buggy that we refuse to support them, and
- must meet all policy requirements presented in this manual.

Furthermore, packages in *contrib* must not require a package in a *non-US* section for compilation or execution.

Examples of packages which would be included in *contrib* or *non-US/contrib* are:
- free packages which require *contrib*, *non-free* packages or packages which are not in our archive at all for compilation or execution, and
- wrapper packages or other sorts of free accessories for non-free programs.

2.1.4 The non-free section

Packages must be placed in *non-free* or *non-US/non-free* if they are not compliant with the DFSG or are encumbered by patents or other legal issues that make their distribution problematic.

In addition, the packages in *non-free* and *non-US/non-free*
- must not be so buggy that we refuse to support them, and
- must meet all policy requirements presented in this manual that it is possible for them to meet.[1]

2.1.5 The non-US sections

Some programs with cryptographic program code need to be stored on the *non-US* server because of United States export restrictions. Such programs must be distributed in the appropriate *non-US* section, either *non-US/main*, *non-US/contrib* or *non-US/non-free*.

[1] It is possible that there are policy requirements which the package is unable to meet, for example, if the source is unavailable. These situations will need to be handled on a case-by-case basis.

This applies only to packages which contain cryptographic code. A package containing a program with an interface to a cryptographic program or a program that's dynamically linked against a cryptographic library should not be distributed via the *non-US* server if it is capable of running without the cryptographic library or program.

2.1.6 Further copyright considerations

Every package must be accompanied by a verbatim copy of its copyright and distribution license in the file `/usr/share/doc/`*`package`*`/copyright` (see 'Copyright information' on page 92 for further details).

We reserve the right to restrict files from being included anywhere in our archives if

- their use or distribution would break a law,
- there is an ethical conflict in their distribution or use,
- we would have to sign a license for them, or
- their distribution would conflict with other project policies.

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; in such a case they must go in *non-free*.

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

Note that under international copyright law (this applies in the United States, too), *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 license terms. However, this can be a politically difficult thing to do and you should ask for advice on the `debian-legal` mailing list first, as explained below.

When in doubt about a copyright, send mail to `<debian-legal@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'.

2.1.7 Subsections

The packages in the sections *main*, *contrib* and *non-free* are grouped further into *subsections* to simplify handling.

The section and subsection for each package should be specified in the package's `Section` control record. However, the maintainer of the Debian archive may override this selection to ensure the consistency of the Debian distribution. The `Section` field should be of the form:

- *subsection* if the package is in the *main* section,
- *section/subsection* if the package is in the *contrib* or *non-free* section, and
- `non-US`, `non-US/contrib` or `non-US/non-free` if the package is in *non-US/main*, *non-US/contrib* or *non-US/non-free* respectively.

The Debian archive maintainers provide the authoritative list of subsections. At present, they are: *admin*, *base*, *comm*, *contrib*, *devel*, *doc*, *editors*, *electronics*, *games*, *graphics*, *hamradio*, *interpreters*, *libs*, *mail*, *math*, *misc*, *net*, *news*, *non-US*, *non-free*, *oldlibs*, *otherosfs*, *science*, *shells*, *sound*, *tex*, *text*, *utils*, *web*, *x11*.

2.2 Priorities

Each package should have a *priority* value, which is included in the package's *control record*. This information is used by the Debian package management tools to separate high-priority packages from less-important packages.

The following *priority levels* are recognised by the Debian package management tools.

`required` Packages which 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 not even be able to use `dpkg` to put things back. Systems with only the `required` packages are probably unusable, 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 say 'What on earth is going on, where is `foo`?', it must be an `important` package.[2] Other packages without which the system will not run well or be usable must also have priority `important`. This does *not* include Emacs, the X Window System, 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 be installed by default if the user doesn't select anything else. It doesn't include many large applications. .

`optional` (In a sense everything that isn't required is optional, but that's not what is meant here.) This is all the software that you might reasonably want to install if you didn't know what it was and don't have specialized requirements. This is a much larger system and includes the X Window System, a full TeX distribution, and many applications. Note that optional packages should not conflict with each other.

[2] This is an important criterion because we are trying to produce, amongst other things, a free Unix.

extra This contains all packages that conflict with others with required, important, standard or optional priorities, or are only likely to be useful if you already know what they are or have specialised requirements.

Packages must not depend on packages with lower priority values (excluding build-time dependencies). In order to ensure this, the priorities of one or more packages may need to be adjusted.

2.3 Binary packages

The Debian GNU/Linux distribution is based on the Debian package management system, called `dpkg`. Thus, all packages in the Debian distribution must be provided in the `.deb` file format.

2.3.1 The package name

Every package must have a name that's unique within the Debian archive.

Package names must consist of lower case letters (`a-z`), digits (`0-9`), plus (`+`) and minus (`-`) signs, and periods (`.`). They must be at least two characters long and must contain at least one letter.

The package name is part of the file name of the `.deb` file and is included in the control field information.

2.3.2 The maintainer of a package

Every package must have a Debian maintainer (the maintainer may be one person or a group of people reachable from a common email address, such as a mailing list). The maintainer is responsible for ensuring that the package is placed in the appropriate distributions.

The maintainer must be specified in the `Maintainer` control field with their correct name and a working email address. If one person maintains several packages, he/she should try to avoid having different forms of their name and email address in the `Maintainer` fields of those packages.

If the maintainer of a package quits from the Debian project, "Debian QA Group" `<packages@qa.debian.org>` takes over the maintainership of the package until someone else volunteers for that task. These packages are called *orphaned packages*.[3]

[3]The detailed procedure for doing this gracefully can be found in the Debian Developer's Reference, either in the `developers-reference` package, or on the Debian FTP server `ftp.debian.org` as `/debian/doc/package-developer/developers-reference.txt.gz` or from the Debian Developer's Reference (`http://www.debian.org/doc/packaging-manuals/developers-reference/`) webpage.

2.3.3 The description of a package

Every Debian package must have an extended description stored in the appropriate field of the control record.

The description should be written so that it gives the system administrator enough information to decide whether to install the package. This description should not just be copied verbatim from the program's documentation. Instructions for configuring or using the package should not be included (that is what installation scripts, manual pages, info files, etc., are for). Copyright statements and other administrivia should not be included either (that is what the copyright file is for).

2.3.4 Dependencies

Every package must specify the dependency information about other packages that are required for the first to work correctly.

For example, a dependency entry must be provided for any shared libraries required by a dynamically-linked executable binary in a package.

Packages are not required to declare any dependencies they have on other packages which are marked `Essential` (see below), and should not do so unless they depend on a particular version of that package.

Sometimes, a package requires another package to be installed *and* configured before it can be installed. In this case, you must specify a `Pre-Depends` entry for the package.

You should not specify a `Pre-Depends` entry for a package before this has been discussed on the `debian-devel` mailing list and a consensus about doing that has been reached.

2.3.5 Virtual packages

Sometimes, there are several packages which offer more-or-less the same functionality. In this case, it's useful to define a *virtual package* whose name describes that common functionality. (The virtual packages only exist logically, not physically; that's why they are called *virtual*.) The packages with this particular function will then *provide* the virtual package. Thus, any other package requiring that function can simply depend on the virtual package without having to specify all possible packages individually.

All packages should use virtual package names where appropriate, and arrange to create new ones if necessary. They should 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.txt` or your local mirror. In addition, it is included in the `debian-policy` package. The procedure for updating the list is described at the top of the file.

2.3.6 Base packages

The packages included in the `base` section have a special function. They form a minimum subset of the Debian GNU/Linux system that is installed before everything else on a new system. Thus, only very few packages are allowed to go into the `base` section to keep the required disk usage very small.

Most of these packages will have the priority value `required` or at least `important`, and many of them will be tagged `essential` (see below).

You must not place any packages into the `base` section before this has been discussed on the `debian-devel` mailing list and a consensus about doing that has been reached.

2.3.7 Essential packages

Some packages are tagged `essential`. (They have `Essential: yes` in their package control record.) This flag is used for packages that are *essential* for a system.

Since these packages cannot be easily removed (one has to specify an extra *force option* to `dpkg` to do so), this flag must not be used unless absolutely necessary. A shared library package must not be tagged `essential`; dependencies will prevent its premature removal, and we need to be able to remove it when it has been superseded.

Since dpkg will not prevent upgrading of other packages while an `essential` package is in an unconfigured state, all `essential` packages must supply all of their core functionality even when unconfigured. If the package cannot satisfy this requirement it must not be tagged as essential, and any packages depending on this package must instead have explicit dependency fields as appropriate.

You must not tag any packages `essential` before this has been discussed on the `debian-devel` mailing list and a consensus about doing that has been reached.

2.3.8 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`.

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

Note that in general 'Scripts' on page 70 applies to package maintainer scripts, too.

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

All packages which supply an instance of a common command name (or, in general, filename) should generally use `update-alternatives`, so that they may be installed together. If `update-alternatives` is not used, then each package must use `Conflicts` to ensure that other packages are de-installed. (In this case, it may be appropriate to specify a conflict against earlier versions of something that previously did not use `update-alternatives`; this is an exception to the usual rule that versioned conflicts should be avoided.)

Prompting in maintainer scripts

Package maintainer scripts may prompt the user if necessary. Prompting may be accomplished by hand, or by communicating with a program, such as `debconf`, which conforms to the Debian Configuration management specification, version 2 or higher. These are included in the `debconf_specification` files in the `debian-policy` package. You may also find this file on the FTP site `ftp.debian.org` in `/debian/doc/package-developer/debconf\_specification.txt.gz` or on your local mirror.[4]

Packages which use the Debian Configuration management specification may contain an additional `config` script and a `templates` file in their control archive. The `config` script might be run before the `preinst` script, and before the package is unpacked or any of its dependancies or pre-dependancies are satisfied. Therefore it must work using only the tools present in *essential* packages.[5]

Packages should try to minimize the amount of prompting they need to do, and they should ensure that the user will only ever 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`), and shared `debconf` variables 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 `config` or `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/share/doc/`*`package`*`/copyright`); neither do instructions on how to use a program (these should be in on-line documentation, where all the users can see them).

[4] 4% of Debian packages [see Debconf stats (`http://kitenet.net/programs/debconf/stats/`)] currently use `debconf` to prompt the user at install time, and this number is growing daily. The benefits of using debconf are briefly explained at Debconf introduction (`http://kitenet.net/doc/debconf-doc/introduction.html`); they include preconfiguration, (mostly) noninteractive installation, elimination of redundant prompting, consistency of user interface, etc. With this increasing number of packages using `debconf`, plus the existance of a nascent second implementation of the Debian configuration management system (`cdebconf`), and the stabalization of the protocol these things use, the time has finally come to reflect the use of these things in policy.

[5] `Debconf` or another tool that implements the Debian Configuration management specification will also be installed, and any versioned dependencies on it will be satisfied before preconfiguration begins.

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

2.4 Source packages

2.4.1 Standards conformance

In the source package's `Standards-Version` control field, you should specify the most recent version number of this policy document with which your package complied when it was last updated. The current version number is 3.5.6.0.

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

The version number has four components: major and minor version number and major and minor patch level. 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 patch level will be changed for any change to the meaning of the standards, however small; the minor patch level will be changed when only cosmetic, typographical or other edits are made which neither change the meaning of the document nor affect the contents of packages.

Thus only the first three components of the policy version are significant in the *Standards-Version* control field, and so either these three components or the all four components may be specified.[6]

You should regularly, and especially if your package has become out of date, check for the newest Policy Manual available and update your package, if necessary. When your package complies with the new standards you should update the `Standards-Version` source package field and release it.[7]

2.4.2 Package relationships

Source packages should specify which binary packages they require to be installed or not to be installed in order to build correctly. For example, if building a package requires a certain compiler, then the compiler should be specified as a build-time dependency.

It is not necessary to explicitly specify build-time relationships on a minimal set of packages that are always needed to compile, link and put in a Debian package a standard "Hello World!" program written

[6] In the past, people specified the full version number in the Standards-Version field, for example '2.3.0.0'. Since minor patch-level changes don't introduce new policy, it was thought it would be better to relax policy and only require the first 3 components to be specified, in this example '2.3.0'. All four components may still be used if someone wishes to do so.

[7] See the file `upgrading-checklist` for information about policy which has changed between different versions of this document.

in C or C++. The required packages are called *build-essential*, and an informational list can be found in `/usr/share/doc/build-essential/list` (which is contained in the `build-essential` package).[8]

When specifying the set of build-time dependencies, one should list only those packages explicitly required by the build. It is not necessary to list packages which are required merely because some other package in the list of build-time dependencies depends on them.[9]

If build-time dependencies are specified, it must be possible to build the package and produce working binaries on a system with only essential and build-essential packages installed and also those required to satisfy the build-time relationships (including any implied relationships). In particular, this means that version clauses should be used rigorously in build-time relationships so that one cannot produce bad or inconsistently configured packages when the relationships are properly satisfied.

2.4.3 Changes to the upstream sources

If changes to the source code are made that are not specific to the needs of the Debian system, they should be sent to the upstream authors in whatever form they prefer so as to be included in the upstream version of the package.

If you need to configure the package differently for Debian or for Linux, and the upstream source doesn't provide a way to do so, you should 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.

You should make sure that the `configure` utility detects the correct architecture specification string (refer to 'Architecture specification strings' on page 79 for details).

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.

[8]Rationale:

- This allows maintaining the list separately from the policy documents (the list does not need the kind of control that the policy documents do).
- Having a separate package allows one to install the build-essential packages on a machine, as well as allowing other packages such as task packages to require installation of the build-essential packages using the depends relation.
- The separate package allows bug reports against the list to be categorized separately from the policy management process in the BTS.

[9]The reason for this is that dependencies change, and you should list all those packages, and *only* those packages that *you* need directly. What others need is their business. For example, if you only link against `libimlib`, you will need to build-depend on `libimlib2-dev` but not against any `libjpeg*` packages, even though `libimlib2-dev` currently depends on them: installation of `libimlib2-dev` will automatically ensure that all of its run-time dependencies are satisfied.

2.4.4 Documenting your changes

You should document your changes and updates to the source package properly in the `debian/changelog` file. (Note that mistakes in changelogs are usually best rectified by making a new changelog entry rather than "rewriting history" by editing old changelog entries.)

In non-experimental packages you must use a format for `debian/changelog` which is supported by the most recent released version of `dpkg`.[10]

2.4.5 Error trapping in makefiles

When `make` invokes a command in a makefile (including your package's upstream makefiles and `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 should include a separate `set -e` command at the start of every makefile command that's actually one of these miniature shell scripts.

2.4.6 Obsolete constructs and libraries

The include file `<varargs.h>` is provided to support end-users compiling very old software; the library `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 patched to use `<stdarg.h>` and `ncurses` instead.

[10] If you wish to use an alternative format, you may do so as long as you include a parser for it in your source package. The parser must have an API compatible with that expected by `dpkg-genchanges` and `dpkg-gencontrol`. If there is general interest in the new format, you should contact the `dpkg` maintainer to have the parser script for it included in the `dpkg` package. (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.)

Chapter 3

Control files and their fields

Many of the tools in the package management suite manipulate data represented in a common format, known as *control data*. The data is often stored in *control files*. Binary and source packages have control files, and the `.changes` files which control the installation of uploaded files are also in control file format. `Dpkg`'s internal databases are in a similar format.

3.1 Syntax of control files

A control file consists of one or more paragraphs of fields. The paragraphs are separated by blank lines. Some control files allow only one paragraph; others allow several, in which case each paragraph usually refers to a different package. (For example, in source packages, the first paragraph refers to the source package, and later paragraphs refer to binary packages generated from the source.)

Each paragraph consists of a series of data fields; each field consists of the field name, followed by a colon and then the data/value associated with that field. It ends at the end of the line. Horizontal whitespace (spaces and tabs) may occur immediately before or after the value and is ignored there; it is conventional to put a single space after the colon. For example, a field might be:

```
Package: libc6
```

the field name is `Package` and the field value `libc6`.

Some fields' values may span several lines; in this case each continuation line *must* start with a space or tab. Any trailing spaces or tabs at the end of individual lines of a field value are ignored.

Except where otherwise stated only a single line of data is allowed and whitespace is not significant in a field body. Whitespace must not appear inside names (of packages, architectures, files or anything else) or version numbers, or between the characters of multi-character version relationships.

Field names are not case-sensitive, but it is usual to capitalize the field names using mixed case as shown below.

Blank lines, or lines consisting only of spaces and tabs, are not allowed within field values or between fields - that would mean a new paragraph.

3.2 List of fields

This list here is not supposed to be exhaustive. Most fields are dealt with elsewhere in this document.

3.2.1 `Package`

The name of the binary package. Package names consist of the alphanumerics and `+ - .` (plus, minus and full stop).

They must be at least two characters long and must start with an alphanumeric character and not be all digits. The use of lowercase package names is strongly recommended unless the package you're building (or referring to, in other fields) is already using uppercase.

3.2.2 `Version`

This lists the source or binary package's version number - see 'Version numbering' on page 19.

3.2.3 `Standards-Version`

The most recent version of the standards (the policy manual and associated texts) with which the package complies. This is updated manually when editing the source package to conform to newer standards; it can sometimes be used to tell when a package needs attention. Its format is described above; see 'Standards conformance' on page 12.

3.2.4 `Distribution`

In a `.changes` file or parsed changelog output this contains the (space-separated) name(s) of the distribution(s) where this version of the package should be installed. Valid distributions are determined by the archive maintainers.[1]

---

[1] Current distribution names are:

stable This is the current 'released' version of Debian GNU/Linux. Once the distribution is *stable* only security fixes and other major bug fixes are allowed. When changes are made to this distribution, the release number is increased (for example: 2.2r1 becomes 2.2r2 then 2.2r3, etc).

unstable This distribution value refers to the *developmental* part of the Debian distribution tree. New packages, new upstream versions of packages and bug fixes go into the *unstable* directory tree. Download from this distribution at your own risk.

testing This distribution value refers to the *testing* part of the Debian distribution tree. It receives its packages from the unstable distribution after a short time lag to ensure that there are no major issues with the unstable packages. It is less prone to breakage than unstable, but still risky. It is not possible to upload packages directly to *testing*.

frozen From time to time, the *testing* distribution enters a state of 'code-freeze' in anticipation of release as a *stable* version. During this period of testing only fixes for existing or newly-discovered bugs will be allowed. The exact details of this stage are determined by the Release Manager.

experimental The packages with this distribution value are deemed by their maintainers to be high risk. Oftentimes they represent early beta or developmental packages from various sources that the maintainers want people to try, but are not ready to be a part of the other parts of the Debian distribution tree. Download at your own risk.

You should list *all* distributions that the package should be installed into.

Chapter 4

Version numbering

Every package has a version number recorded in its `Version` control file field.

The package management system imposes an ordering on version numbers, so that it can tell whether packages are being up- or downgraded and so that package system front end applications can tell whether a package it finds available is newer than the one installed on the system. The version number format has the most significant parts (as far as comparison is concerned) at the beginning.

The version number format is: [*epoch*`:`]*upstream_version*[-*debian_revision*]

The three components here are:

epoch This is a single (generally small) unsigned integer. It may be omitted, in which case zero is assumed. If it is omitted then the *upstream_version* may not contain any colons.

It is provided to allow mistakes in the version numbers of older versions of a package, and also a package's previous version numbering schemes, to be left behind.

upstream_version This is the main part of the version number. It is usually the version number of the original ('upstream') package from which the `.deb` file has been made, if this is applicable. Usually this will be in the same format as that specified by the upstream author(s); however, it may need to be reformatted to fit into the package management system's format and comparison scheme.

The comparison behavior of the package management system with respect to the *upstream_version* is described below. The *upstream_version* portion of the version number is mandatory.

The *upstream_version* may contain only alphanumerics[1] and the characters `. + - :` (full stop, plus, hyphen, colon) and should start with a digit. If there is no *debian_revision* then hyphens are not allowed; if there is no *epoch* then colons are not allowed.

[1] Alphanumerics are `A-Za-z0-9` only.

debian_revision This part of the version number specifies the version of the Debian package based on the upstream version. It may contain only alphanumerics and the characters + and `.` (plus and full stop) and is compared in the same way as the *upstream_version* is.

It is optional; if it isn't present then the *upstream_version* may not contain a hyphen. This format represents the case where a piece of software was written specifically to be turned into a Debian package, and so there is only one 'debianization' of it and therefore no revision indication is required.

It is conventional to restart the *debian_revision* at `1` each time the *upstream_version* is increased.

The package management system will break the version number apart at the last hyphen in the string (if there is one) to determine the *upstream_version* and *debian_revision*. The absence of a *debian_revision* compares earlier than the presence of one (but note that the *debian_revision* is the least significant part of the version number).

The *upstream_version* and *debian_revision* parts are compared by the package management system using the same algorithm:

The strings are compared from left to right.

First the initial part of each string consisting entirely of non-digit characters is determined. These two parts (one of which may be empty) are compared lexically. If a difference is found it is returned. The lexical comparison is a comparison of ASCII values modified so that all the letters sort earlier than all the non-letters.

Then the initial part of the remainder of each string which consists entirely of digit characters is determined. The numerical values of these two parts are compared, and any difference found is returned as the result of the comparison. For these purposes an empty string (which can only occur at the end of one or both version strings being compared) counts as zero.

These two steps (comparing and removing initial non-digit strings and initial digit strings) are repeated until a difference is found or both strings are exhausted.

Note that the purpose of epochs is to allow us to leave behind mistakes in version numbering, and to cope with situations where the version numbering scheme changes. It is *not* intended to cope with version numbers containing strings of letters which the package management system cannot interpret (such as `ALPHA` or `pre-`), or with silly orderings (the author of this manual has heard of a package whose versions went `1.1`, `1.2`, `1.3`, `1`, `2.1`, `2.2`, `2` and so forth).

If an upstream package has problematic version numbers they should be converted to a sane form for use in the `Version` field.

4.1 Version numbers based on dates

In general, Debian packages should use the same version numbers as the upstream sources.

However, in some cases where the upstream version number is based on a date (e.g., a development 'snapshot' release) the package management system cannot handle these version numbers without epochs. For example, dpkg will consider '96May01' to be greater than '96Dec24'.

To prevent having to use epochs for every new upstream version, the version number should be changed to the following format in such cases: '19960501', '19961224'. It is up to the maintainer whether he/she wants to bother the upstream maintainer to change the version numbers upstream, too.

Note that other version formats based on dates which are parsed correctly by the package management system should *not* be changed.

Native Debian packages (i.e., packages which have been written especially for Debian) whose version numbers include dates should always use the 'YYYYMMDD' format.

Chapter 5

Packaging Considerations

5.1 Time Stamps

Maintainers should preserve the modification times of the upstream source files in a package, as far as is reasonably possible.[1]

5.2 `debian/rules` - the main building script

This file must be an executable makefile, and contains the package-specific recipes for compiling the package and building binary package(s) from the source.

It must start with the line `#!/usr/bin/make -f`, so that it can be invoked by saying its name rather than invoking `make` explicitly.

Since an interactive `debian/rules` script makes it impossible to auto-compile that package and also makes it hard for other people to reproduce the same binary package, all *required targets* MUST be non-interactive. At a minimum, required targets are the ones called by `dpkg-buildpackage`, namely, *clean*, *binary*, *binary-arch*, *binary-indep*, and *build*. It also follows that any target that these targets depend on must also be non-interactive.

The required and optional targets are as follows:

`build`, `build-arch` (optional), `build-indep` (optional) The `build` target should perform all non-interactive configuration and compilation of the package. If a package has an interactive pre-build configuration routine, the Debianized source package must either be built after this has taken place

[1] The rationale is that there is some information conveyed by knowing the age of the file, for example, you could recognize that some documentation is very old by looking at the modification time, so it would be nice if the modification time of the upstream source would be preserved.

(so that the binary package can be built without rerunning the configuration) or the configuration routine modified to become non-interactive. (The latter is preferable if there are architecture-specific features detected by the configuration routine.)

For some packages, notably ones where the same source tree is compiled in different ways to produce two binary packages, the `build` target does not make much sense. For these packages it is good enough to provide two (or more) targets (`build-a` and `build-b` or whatever) for each of the ways of building the package, and a `build` target that does nothing. The `binary` target will have to build the package in each of the possible ways and make the binary package out of each.

The `build` target must not do anything that might require root privilege.

The `build` target may need to run the `clean` target first - see below.

When a package has a configuration and build routine which takes a long time, or when the makefiles are poorly designed, or when `build` needs to run `clean` first, it is a good idea to `touch build` when the build process is complete. This will ensure that if `debian/rules build` is run again it will not rebuild the whole program.[2]

`binary, binary-arch, binary-indep` The `binary` target must be all that is necessary for the user to build the binary package(s) produced from this source package. All of these targets are required to be non-interactive. It is split into two parts: `binary-arch` builds the binary packages which are specific to a particular architecture, and `binary-indep` builds those which are not.

`binary` may be (and commonly is) a target with no commands which simply depends on `binary-arch` and `binary-indep`.

Both `binary-*` targets should depend on the `build` target, or on the appropriate `build-arch` or `build-indep` target, if provided, so that the package is built if it has not been already. It should then create the relevant binary package(s), using `dpkg-gencontrol` to make their control files and `dpkg-deb` to build them and place them in the parent of the top level directory.

Both the `binary-arch` and `binary-indep` targets *must* exist. If one of them has nothing to do (which will always be the case if the source generates only a single binary package, whether architecture-dependent or not), it must still exist and must always succeed.

The `binary` targets must be invoked as root.[3]

`clean` This must undo any effects that the `build` and `binary` targets may have had, except that it should leave alone any output files created in the parent directory by a run of a `binary` target. This target must be non-interactive.

If a `build` file is touched at the end of the `build` target, as suggested above, it should be removed as the first action that `clean` performs, so that running `build` again after an interrupted `clean` doesn't think that everything is already done.

[2]Another common way to do this is for `build` to depend on `build-stamp` and to do nothing else, and for the `build-stamp` target to do the building and to `touch build-stamp` on completion. This is especially useful if the build routine creates a file or directory called `build`; in such a case, `build` will need to be listed as a phony target (i.e., as a dependency of the `.PHONY` target). See the documentation of `make` for more information on phony targets.

[3]The `fakeroot` package often allows one to build a package correctly even without being root.

The `clean` target may need to be invoked as root if `binary` has been invoked since the last `clean`, or if `build` has been invoked as root (since `build` may create directories, for example).

`get-orig-source` (optional) This target fetches the most recent version of the original source package from a canonical archive site (via FTP or WWW, for example), does any necessary rearrangement to turn it into the original source tar file format described below, and leaves it in the current directory.

This target may be invoked in any directory, and should take care to clean up any temporary files it may have left.

This target is optional, but providing it if possible is a good idea.

The `build`, `binary` and `clean` targets must be invoked with the current directory being the package's top-level directory.

Additional targets may exist in `debian/rules`, either as published or undocumented interfaces or for the package's internal use.

The architectures we build on and build for are determined by `make` variables using the utility `dpkg-architecture`. You can determine the Debian architecture and the GNU style architecture specification string for the build machine (the machine type we are building on) as well as for the host machine (the machine type we are building for). Here is a list of supported `make` variables:

- `DEB_*_ARCH` (the Debian architecture)
- `DEB_*_GNU_TYPE` (the GNU style architecture specification string)
- `DEB_*_GNU_CPU` (the CPU part of `DEB_*_GNU_TYPE`)
- `DEB_*_GNU_SYSTEM` (the System part of `DEB_*_GNU_TYPE`)

where `*` is either `BUILD` for specification of the build machine or `HOST` for specification of the host machine.

Backward compatibility can be provided in the rules file by setting the needed variables to suitable default values; please refer to the documentation of `dpkg-architecture` for details.

It is important to understand that the `DEB_*_ARCH` string only determines which Debian architecture we are building on or for. It should not be used to get the CPU or system information; the GNU style variables should be used for that.

5.3 `debian/changelog`

This file records the changes to the Debian-specific parts of the package[4].

It has a special format which allows the package building tools to discover which version of the package is being built and find out other release-specific information.

[4]Though there is nothing stopping an author who is also the Debian maintainer from using it for all their changes, it will have to be renamed if the Debian and upstream maintainers become different people. In such a case, however, it might be better to maintain the package as a non-native package.

That format is a series of entries like this:

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

  * change details
    more change details

  * even more change details

     -- maintainer name <email
       address>[two spaces]  date
```

package and *version* are the source package name and version number.

distribution(s) lists the distributions where this version should be installed when it is uploaded - it is copied to the `Distribution` field in the `.changes` file. See ‘`Distribution`’ on page 16.

urgency is the value for the `Urgency` field in the `.changes` file for the upload. It is not possible to specify an urgency containing commas; commas are used to separate `keyword=value` settings in the `dpkg` changelog format (though there is currently only one useful *keyword*, `urgency`).[5]

The change details may in fact be any series of lines starting with at least two spaces, but conventionally each change starts with an asterisk and a separating space and continuation lines are indented so as to bring them in line with the start of the text above. Blank lines may be used here to separate groups of changes, if desired.

If this upload resolves bugs recorded in the Bug Tracking System (BTS), they may be automatically closed on the inclusion of this package into the Debian archive by including the string: `closes:  Bug#nnnnn` in the change details.[6]

The maintainer name and email address used in the changelog should be the details of the person uploading *this* version. They are *not* necessarily those of the usual package maintainer. The information here will be copied to the `Changed-By` field in the `.changes` file, and then later used to send an acknowledgement when the upload has been installed.

The *date* should be in RFC822 format[7]; it should include the time zone specified numerically, with the time zone name or abbreviation optionally present as a comment in parentheses.

[5] Recognised urgency values are `low`, `medium`, `high` and `emergency`. They have an effect on how quickly a package will be considered for inclusion into the `testing` distribution, and give an indication of the importance of any fixes included in this upload.

[6] To be precise, the string should match the following Perl regular expression:

```
/closes:\s*(?:bug)?\#?\s?\d+(?:,\s*(?:bug)?\#?\s?\d+)*/i
```

Then all of the bug numbers listed will be closed by the archive maintenance script (`katie`), or in the case of an NMU, marked as fixed.

[7] This is generated by the `822-date` program.

The first 'title' line with the package name should start at the left hand margin; the 'trailer' line with the maintainer and date details should be preceded by exactly one space. The maintainer details and the date must be separated by exactly two spaces.

5.3.1 Defining alternative changelog formats

It is possible to use a different format to the standard one, by providing a parser for the format you wish to use.

A changelog parser must not interact with the user at all.

5.4 `debian/substvars` and variable substitutions

When `dpkg-gencontrol`, `dpkg-genchanges` and `dpkg-source` generate control files they perform variable substitutions on their output just before writing it. Variable substitutions have the form `${`*`variable`*`}`. The optional file `debian/substvars` contains variable substitutions to be used; variables can also be set directly from `debian/rules` using the `-V` option to the source packaging commands, and certain predefined variables are also available.

The `debian/substvars` file is usually generated and modified dynamically by `debian/rules` targets; in this case it must be removed by the `clean` target.

See `dpkg-source(1)` for full details about source variable substitutions, including the format of `debian/substvars`.

5.5 `debian/files`

This file is not a permanent part of the source tree; it is used while building packages to record which files are being generated. `dpkg-genchanges` uses it when it generates a `.changes` file.

It should not exist in a shipped source package, and so it (and any backup files or temporary files such as `files.new`[8]) should be removed by the `clean` target. It may also be wise to ensure a fresh start by emptying or removing it at the start of the `binary` target.

When `dpkg-gencontrol` is run for a binary package, it adds an entry to `debian/files` for the `.deb` file that will be created when `dpkg-deb --build` is run for that binary package. So for most packages all that needs to be done with this file is to delete it in the `clean` target.

[8] `files.new` is used as a temporary file by `dpkg-gencontrol` and `dpkg-distaddfile` - they write a new version of `files` here before renaming it, to avoid leaving a corrupted copy if an error occurs

If a package upload includes files besides the source package and any binary packages whose control files were made with `dpkg-gencontrol` then they should be placed in the parent of the package's top-level directory and `dpkg-distaddfile` should be called to add the file to the list in `debian/files`.

5.6 Restrictions on objects in source packages

The source package may not contain any hard links[9], device special files, sockets or setuid or setgid files.[10]

5.7 Descriptions of packages - the `Description` field

The description is intended to describe the program to a user who has never met it before so that they know whether they want to install it. It should also give information about the significant dependencies and conflicts between this package and others, so that the user knows why these dependencies and conflicts have been declared.

5.7.1 Notes about writing descriptions

The single line synopsis should be kept brief - certainly under 80 characters.

Do not include the package name in the synopsis line. The display software knows how to display this already, and you do not need to state it. Remember that in many situations the user may only see the synopsis line - make it as informative as you can.

Do not try to continue the single line synopsis into the extended description. This will not work correctly when the full description is displayed, and makes no sense where only the summary (the single line synopsis) is available.

The extended description should describe what the package does and how it relates to the rest of the system (in terms of, for example, which subsystem it is which part of).

The description field needs to make sense to anyone, even people who have no idea about any of the things the package deals with.[11]

Put important information first, both in the synopsis and extended description. Sometimes only the first part of the synopsis or of the description will be displayed. You can assume that there will usually be a way to see the whole extended description.

---

[9] This is not currently detected when building source packages, but only when extracting them. Hard links may be permitted at some point in the future, but would require a fair amount of work.

[10] Setgid directories are allowed.

[11] The blurb that comes with a program in its announcements and/or `README` files is rarely suitable for use in a description. It is usually aimed at people who are already in the community where the package is used.

You may include information about dependencies and so forth in the extended description, if you wish.

Do not use tab characters. Their effect is not predictable.

Chapter 6

Package maintainer scripts and installation procedure

6.1 Introduction to package maintainer scripts

It is possible to supply scripts as part of a package which the package management system will run for you when your package is installed, upgraded or removed.

These scripts are the files `preinst`, `postinst`, `prerm` and `postrm` in the control area of the package. They must be proper executable files; if they are scripts (which is recommended), they must start with the usual `#!` convention. They should be readable and executable by anyone, and not world-writable.

The package management system looks at the exit status from these scripts. It is important that they exit with a non-zero status if there is an error, so that the package management system can stop its processing. For shell scripts this means that you *almost always* need to use `set -e` (this is usually true when writing shell scripts, in fact). It is also important, of course, that they don't exit with a non-zero status if everything went well.

When a package is upgraded a combination of the scripts from the old and new packages is called during the upgrade procedure. If your scripts are going to be at all complicated you need to be aware of this, and may need to check the arguments to your scripts.

Broadly speaking the `preinst` is called before (a particular version of) a package is installed, and the `postinst` afterwards; the `prerm` before (a version of) a package is removed and the `postrm` afterwards.

Programs called from maintainer scripts should not normally have a path prepended to them. Before installation is started, the package management system checks to see if the programs `ldconfig`, `start-stop-daemon`, `install-info`, and `update-rc.d` can be found via the `PATH` environment variable. Those programs, and any other program that one would expect to be on the `PATH`, should thus be invoked without an absolute pathname. Maintainer scripts should also not reset the `PATH`, though they might choose to modify it by prepending or appending package-specific directories. These considerations really apply to all shell scripts.

6.2 Maintainer scripts Idempotency

It is necessary for the error recovery procedures that the scripts be idempotent. This means that if it is run successfully, and then it is called again, it doesn't bomb out or cause any harm, but just ensures that everything is the way it ought to be. If the first call failed, or aborted half way through for some reason, the second call should merely do the things that were left undone the first time, if any, and exit with a success status if everything is OK.[1]

6.3 Controlling terminal for maintainer scripts

The maintainer scripts are guaranteed to run with a controlling terminal and can interact with the user. If they need to prompt for passwords, do full-screen interaction or something similar you should do these things to and from `/dev/tty`, since `dpkg` will at some point redirect scripts' standard input and output so that it can log the installation process. Likewise, because these scripts may be executed with standard output redirected into a pipe for logging purposes, Perl scripts should set unbuffered output by setting `$|=1` so that the output is printed immediately rather than being buffered.

Each script should return a zero exit status for success, or a nonzero one for failure.

6.4 Summary of ways maintainer scripts are called

- *new-preinst* `install`
- *new-preinst* `install` *old-version*
- *new-preinst* `upgrade` *old-version*
- *old-preinst* `abort-upgrade` *new-version*
- *postinst* `configure` *most-recently-configured-version*
- *old-postinst* `abort-upgrade` *new-version*
- *conflictor's-postinst* `abort-remove` `in-favour` *package new-version*
- *deconfigured's-postinst* `abort-deconfigure` `in-favour` *failed-install-package version* `removing` *conflicting-package version*
- *prerm* `remove`
- *old-prerm* `upgrade` *new-version*
- *new-prerm* `failed-upgrade` *old-version*

[1]This is so that if an error occurs, the user interrupts `dpkg` or some other unforeseen circumstance happens you don't leave the user with a badly-broken package when `dpkg` attempts to repeat the action.

- *conflictor's-prerm* `remove` `in-favour` *package new-version*
- *deconfigured's-prerm* `deconfigure` `in-favour` *package-being-installed version* `removing` *conflicting-package version*
- *postrm* `remove`
- *postrm* `purge`
- *old-postrm* `upgrade` *new-version*
- *new-postrm* `failed-upgrade` *old-version*
- *new-postrm* `abort-install`
- *new-postrm* `abort-install` *old-version*
- *new-postrm* `abort-upgrade` *old-version*
- *disappearer's-postrm* `disappear` *overwriter overwriter-version*

6.5 Details of unpack phase of installation or upgrade

The procedure on installation/upgrade/overwrite/disappear (i.e., when running `dpkg --unpack`, or the unpack stage of `dpkg --install`) is as follows. In each case, if a major error occurs (unless listed below) the actions are, in general, run backwards - this means that the maintainer scripts are run with different arguments in reverse order. These are the 'error unwind' calls listed below.

1. (a) If a version of the package is already installed, call

   ```
   old-prerm upgrade new-version
   ```

   (b) If the script runs but exits with a non-zero exit status, `dpkg` will attempt:

   ```
   new-prerm failed-upgrade old-version
   ```

   Error unwind, for both the above cases:

   ```
   old-postinst abort-upgrade new-version
   ```

2. If a 'conflicting' package is being removed at the same time:

   (a) If any packages depended on that conflicting package and `--auto-deconfigure` is specified, call, for each such package:

   ```
   deconfigured's-prerm deconfigure \
     in-favour package-being-installed version \
       removing conflicting-package version
   ```

   Error unwind:

   ```
   deconfigured's-postinst abort-deconfigure \
     in-favour package-being-installed-but-failed version \
       removing conflicting-package version
   ```

The deconfigured packages are marked as requiring configuration, so that if `--install` is used they will be configured again if possible.

(b) To prepare for removal of the conflicting package, call:

```
conflictor's-prerm remove \
  in-favour package new-version
```

Error unwind:

```
conflictor's-postinst abort-remove \
  in-favour package new-version
```

3. (a) If the package is being upgraded, call:

```
new-preinst upgrade old-version
```

(b) Otherwise, if the package had some configuration files from a previous version installed (i.e., it is in the 'configuration files only' state):

```
new-preinst install old-version
```

(c) Otherwise (i.e., the package was completely purged):

```
new-preinst install
```

Error unwind actions, respectively:

```
new-postrm abort-upgrade old-version
new-postrm abort-install old-version
new-postrm abort-install
```

4. The new package's files are unpacked, overwriting any that may be on the system already, for example any from the old version of the same package or from another package. Backups of the old files are kept temporarily, and if anything goes wrong the package management system will attempt to put them back as part of the error unwind.

   It is an error for a package to contains files which are on the system in another package, unless `Replaces` is used (see 'Overwriting files and replacing packages - `Replaces`' on page 41).

   It is a more serious error for a package to contain a plain file or other kind of non-directory where another package has a directory (again, unless `Replaces` is used). This error can be overridden if desired using `--force-overwrite-dir`, but this is not advisable.

   Packages which overwrite each other's files produce behavior which, though deterministic, is hard for the system administrator to understand. It can easily lead to 'missing' programs if, for example, a package is installed which overwrites a file from another package, and is then removed again.[2]

   A directory will never be replaced by a symbolic link to a directory or vice versa; instead, the existing state (symlink or not) will be left alone and `dpkg` will follow the symlink if there is one.

[2]Part of the problem is due to what is arguably a bug in `dpkg`.

5. (a) If the package is being upgraded, call

    ```
    old-postrm upgrade new-version
    ```

   (b) If this fails, `dpkg` will attempt:

    ```
    new-postrm failed-upgrade old-version
    ```

    Error unwind, for both cases:

    ```
    old-preinst abort-upgrade new-version
    ```

   This is the point of no return - if `dpkg` gets this far, it won't back off past this point if an error occurs. This will leave the package in a fairly bad state, which will require a successful re-installation to clear up, but it's when `dpkg` starts doing things that are irreversible.

6. Any files which were in the old version of the package but not in the new are removed.

7. The new file list replaces the old.

8. The new maintainer scripts replace the old.

9. Any packages all of whose files have been overwritten during the installation, and which aren't required for dependencies, are considered to have been removed. For each such package

   (a) `dpkg` calls:

    ```
    disappearer's-postrm disappear \
      overwriter overwriter-version
    ```

   (b) The package's maintainer scripts are removed.

   (c) It is noted in the status database as being in a sane state, namely not installed (any conffiles it may have are ignored, rather than being removed by `dpkg`). Note that disappearing packages do not have their prerm called, because `dpkg` doesn't know in advance that the package is going to vanish.

10. Any files in the package we're unpacking that are also listed in the file lists of other packages are removed from those lists. (This will lobotomize the file list of the 'conflicting' package if there is one.)

11. The backup files made during installation, above, are deleted.

12. The new package's status is now sane, and recorded as 'unpacked'.

    Here is another point of no return - if the conflicting package's removal fails we do not unwind the rest of the installation; the conflicting package is left in a half-removed limbo.

13. If there was a conflicting package we go and do the removal actions (described below), starting with the removal of the conflicting package's files (any that are also in the package being installed have already been removed from the conflicting package's file list, and so do not get removed now).

6.6 Details of configuration

When we configure a package (this happens with `dpkg --install` and `dpkg --configure`), we first update any `conffiles` and then call:

```
postinst configure most-recently-configured-version
```

No attempt is made to unwind after errors during configuration.

If there is no most recently configured version `dpkg` will pass a null argument; older versions of dpkg may pass `<unknown>` (including the angle brackets) in this case. Even older ones do not pass a second argument at all, under any circumstances.

6.7 Details of removal and/or configuration purging

1. ```
   prerm remove
   ```
2. The package's files are removed (except `conffiles`).
3. ```
   postrm remove
   ```
4. All the maintainer scripts except the `postrm` are removed.

   If we aren't purging the package we stop here. Note that packages which have no `postrm` and no `conffiles` are automatically purged when removed, as there is no difference except for the `dpkg` status.
5. The `conffiles` and any backup files (`~`-files, `#*#` files, `%`-files, `.dpkg-{old,new,tmp}`, etc.) are removed.
6. ```
   postrm purge
   ```
7. The package's file list is removed.

No attempt is made to unwind after errors during removal.
   ```

Chapter 7

Declaring relationships between packages

Packages can declare in their control file that they have certain relationships to other packages - for example, that they may not be installed at the same time as certain other packages, and/or that they depend on the presence of others, or that they should overwrite files in certain other packages if present.

This is done using the `Depends`, `Pre-Depends`, `Recommends`, `Suggests`, `Enhances`, `Conflicts`, `Provides` and `Replaces` control file fields.

Source packages may declare relationships to binary packages, saying that they require certain binary packages to be installed or absent at the time of building the package.

This is done using the `Build-Depends`, `Build-Depends-Indep`, `Build-Conflicts` and `Build-Conflicts-Indep` control file fields.

7.1 Syntax of relationship fields

These fields all have a uniform syntax. They are a list of package names separated by commas.

In the `Depends`, `Recommends`, `Suggests`, `Pre-Depends`, `Build-Depends` and `Build-Depends-Indep` control file fields of the package, which declare dependencies on other packages, the package names listed may also include lists of alternative package names, separated by vertical bar (pipe) symbols `|`. In such a case, if any one of the alternative packages is installed, that part of the dependency is considered to be satisfied.

All of the fields except for `Provides` may restrict their applicability to particular versions of each named package. This is done in parentheses after each individual package name; the parentheses should contain a relation from the list below followed by a version number, in the format described in 'Version numbering' on page 19.

The relations allowed are <<, <=, =, >= and >> for strictly earlier, earlier or equal, exactly equal, later or equal and strictly later, respectively. The deprecated forms < and > were used to mean earlier/later or equal, rather than strictly earlier/later, so they should not appear in new packages (though `dpkg` still supports them).

Whitespace may appear at any point in the version specification subject to the rules in 'Syntax of control files' on page 15, and must appear where it's necessary to disambiguate; it is not otherwise significant. For consistency and in case of future changes to `dpkg` it is recommended that a single space be used after a version relationship and before a version number; it is also conventional to put a single space after each comma, on either side of each vertical bar, and before each open parenthesis.

For example, a list of dependencies might appear as:

```
Package: mutt
Version: 1.3.17-1
Depends: libc6 (>= 2.2.1), exim | mail-transport-agent
```

All fields that specify build-time relationships (`Build-Depends`, `Build-Depends-Indep`, `Build-Conflicts` and `Build-Conflicts-Indep`) may be restricted to a certain set of architectures. This is indicated in brackets after each individual package name and the optional version specification. The brackets enclose a list of Debian architecture names separated by whitespace. Exclamation marks may be prepended to each of the names. (It is not permitted for some names to be prepended with exclamation marks and others not.) If the current Debian host architecture is not in this list and there are no exclamation marks in the list, or it is in the list with a prepended exclamation mark, the package name and the associated version specification are ignored completely for the purposes of defining the relationships.

For example:

```
Source: glibc
Build-Depends-Indep: texinfo
Build-Depends: kernel-headers-2.2.10 [!hurd-i386],
  hurd-dev [hurd-i386], gnumach-dev [hurd-i386]
```

Note that the binary package relationship fields such as `Depends` appear in one of the binary package sections of the control file, whereas the build-time relationships such as `Build-Depends` appear in the source package section of the control file (which is the first section).

7.2 Binary Dependencies - `Depends`, `Recommends`, `Suggests`, `Enhances`, `Pre-Depends`

These five fields are used to declare a dependency relationship by one package on another. Except for `Enhances`, they appear in the depending (binary) package's control file. (`Enhances` appears in the recommending package's control file.)

A `Depends` field takes effect *only* when a package is to be configured. It does not prevent a package being on the system in an unconfigured state while its dependencies are unsatisfied, and it is possible to replace a package whose dependencies are satisfied and which is properly installed with a different version whose dependencies are not and cannot be satisfied; when this is done the depending package will be left unconfigured (since attempts to configure it will give errors) and will not function properly. If it is necessary, a `Pre-Depends` field can be used, which has a partial effect even when a package is being unpacked, as explained in detail below. (The other three dependency fields, `Recommends`, `Suggests` and `Enhances`, are only used by the various front-ends to `dpkg` such as `dselect`.)

For this reason packages in an installation run are usually all unpacked first and all configured later; this gives later versions of packages with dependencies on later versions of other packages the opportunity to have their dependencies satisfied.

The `Depends` field thus allows package maintainers to impose an order in which packages should be configured.

The meaning of the five dependency fields is as follows:

`Depends` This declares an absolute dependency. A package will not be configured unless all of the packages listed in its `Depends` field have been correctly configured.

The `Depends` field should be used if the depended-on package is required for the depending package to provide a significant amount of functionality.

The `Depends` field should also be used if the `postinst`, `prerm` or `postrm` scripts require the package to be present in order to run. Note, however, that the `postrm` cannot rely on any non-essential packages to be present during the `purge` phase.

`Recommends` This declares a strong, but not absolute, dependency.

The `Recommends` field should list packages that would be found together with this one in all but unusual installations.

`Suggests` This is used to declare that one package may be more useful with one or more others. Using this field tells the packaging system and the user that the listed packages are related to this one and can perhaps enhance its usefulness, but that installing this one without them is perfectly reasonable.

`Enhances` This field is similar to Suggests but works in the opposite direction. It is used to declare that a package can enhance the functionality of another package.

`Pre-Depends` This field is like `Depends`, except that it also forces `dpkg` to complete installation of the packages named before even starting the installation of the package which declares the pre-dependency, as follows:

When a package declaring a pre-dependency is about to be *unpacked* the pre-dependency can be satisfied if the depended-on package is either fully configured, *or even if* the depended-on package(s) are only unpacked or half-configured, provided that they have been configured correctly at some point in the past (and not removed or partially removed since). In this case, both the previously-configured and currently unpacked or half-configured versions must satisfy any version clause in the `Pre-Depends` field.

When the package declaring a pre-dependency is about to be *configured*, the pre-dependency will be treated as a normal `Depends`, that is, it will be considered satisfied only if the depended-on package has been correctly configured.

`Pre-Depends` should be used sparingly, preferably only by packages whose premature upgrade or installation would hamper the ability of the system to continue with any upgrade that might be in progress.

`Pre-Depends` are also required if the `preinst` script depends on the named package. It is best to avoid this situation if possible.

When selecting which level of dependency to use you should consider how important the depended-on package is to the functionality of the one declaring the dependency. Some packages are composed of components of varying degrees of importance. Such a package should list using `Depends` the package(s) which are required by the more important components. The other components' requirements may be mentioned as Suggestions or Recommendations, as appropriate to the components' relative importance.

7.3 Conflicting binary packages - `Conflicts`

When one binary package declares a conflict with another using a `Conflicts` field, `dpkg` will refuse to allow them to be installed on the system at the same time.

If one package is to be installed, the other must be removed first - if the package being installed is marked as replacing (see 'Overwriting files and replacing packages - `Replaces`' on the next page) the one on the system, or the one on the system is marked as deselected, or both packages are marked `Essential`, then `dpkg` will automatically remove the package which is causing the conflict, otherwise it will halt the installation of the new package with an error. This mechanism is specifically designed to produce an error when the installed package is `Essential`, but the new package is not.

A package will not cause a conflict merely because its configuration files are still installed; it must be at least half-installed.

A special exception is made for packages which declare a conflict with their own package name, or with a virtual package which they provide (see below): this does not prevent their installation, and allows a package to conflict with others providing a replacement for it. You use this feature when you want the package in question to be the only package providing some feature.

A `Conflicts` entry should almost never have an 'earlier than' version clause. This would prevent `dpkg` from upgrading or installing the package which declared such a conflict until the upgrade or removal of the conflicted-with package had been completed.

7.4 Virtual packages - `Provides`

As well as the names of actual ('concrete') packages, the package relationship fields `Depends`, `Recommends`, `Suggests`, `Enhances`, `Pre-Depends`, `Conflicts`, `Build-Depends`, `Build-Depends-Indep`, `Build-Conflicts` and `Build-Conflicts-Indep` may mention 'virtual packages'.

A *virtual package* is one which appears in the `Provides` control file field of another package. The effect is as if the package(s) which provide a particular virtual package name had been listed by name everywhere the virtual package name appears.

If there are both a real and a virtual package of the same name then the dependency may be satisfied (or the conflict caused) by either the real package or any of the virtual packages which provide it. This is so that, for example, supposing we have

```
Package: foo
Depends: bar
```

and someone else releases an enhanced version of the `bar` package (for example, a non-US variant), they can say:

```
Package: bar-plus
Provides: bar
```

and the `bar-plus` package will now also satisfy the dependency for the `foo` package.

If a dependency or a conflict has a version number attached then only real packages will be considered to see whether the relationship is satisfied (or the prohibition violated, for a conflict) - it is assumed that a real package which provides the virtual package is not of the 'right' version. So, a `Provides` field may not contain version numbers, and the version number of the concrete package which provides a particular virtual package will not be looked at when considering a dependency on or conflict with the virtual package name.

It is likely that the ability will be added in a future release of `dpkg` to specify a version number for each virtual package it provides. This feature is not yet present, however, and is expected to be used only infrequently.

If you want to specify which of a set of real packages should be the default to satisfy a particular dependency on a virtual package, you should list the real package as an alternative before the virtual one.

7.5 Overwriting files and replacing packages - `Replaces`

The `Replaces` control file field has two distinct purposes, which come into play in different situations.

7.5.1 Overwriting files in other packages

Firstly, as mentioned before, it is usually an error for a package to contain files which are on the system in another package.

However, if the overwriting package declares that it `Replaces` the one containing the file being overwritten, then `dpkg` will replace the file from the old package with that from the new. The file will no longer be listed as 'owned' by the old package.

If a package is completely replaced in this way, so that `dpkg` does not know of any files it still contains, it is considered to have 'disappeared'. It will be marked as not wanted on the system (selected for removal) and not installed. Any `conffiles` details noted for the package will be ignored, as they will have been taken over by the overwriting package. The package's `postrm` script will be run with a special argument to allow the package to do any final cleanup required. See 'Summary of ways maintainer scripts are called' on page 32.

If an installed package, `foo` say, declares that it replaces another, `bar`, and an attempt is made to install `bar`, `dpkg` will discard files in the `bar` package which would overwrite those already present in `foo`. This is so that you can install an older version of a package without problems.

For this usage of `Replaces`, virtual packages (see 'Virtual packages - `Provides`' on the page before) are not considered when looking at a `Replaces` field - the packages declared as being replaced must be mentioned by their real names.

Furthermore, this usage of `Replaces` only takes effect when both packages are at least partially on the system at once, so that it can only happen if they do not conflict or if the conflict has been overridden.

7.5.2 Replacing whole packages, forcing their removal

Secondly, `Replaces` allows the packaging system to resolve which package should be removed when there is a conflict - see 'Conflicting binary packages - `Conflicts`' on page 40. This usage only takes effect when the two packages *do* conflict, so that the two usages of this field do not interfere with each other.

In this situation, the package declared as being replaced can be a virtual package, so for example, all mail transport agents (MTAs) would have the following fields in their control files:

```
Provides: mail-transport-agent
Conflicts: mail-transport-agent
Replaces: mail-transport-agent
```

ensuring that only one MTA can be installed at any one time.

7.6 Relationships between source and binary packages - `Build-Depends`, `Build-Depends-Indep`, `Build-Conflicts`, `Build-Conflicts-Indep`

A source package may declare a dependency or a conflict on a binary package, indicating which packages are required to be present on the system in order to build the binary packages from the source package. This is done with the control file fields `Build-Depends`, `Build-Depends-Indep`, `Build-Conflicts` and `Build-Conflicts-Indep`. The dependencies and conflicts they define must be satisfied (as defined earlier for binary packages) in order to invoke the targets in `debian/rules`, as follows:

`Build-Depends`, `Build-Conflicts` The `Build-Depends` and `Build-Conflicts` fields must be satisfied when any of the following targets is invoked: `build`, `binary`, `binary-arch` and `binary-indep`.

`Build-Depends-Indep`, `Build-Conflicts-Indep` The `Build-Depends-Indep` and `Build-Conflicts-Indep` fields must be satisfied when any of the following targets is invoked: `binary` and `binary-indep`.

Chapter 8

Configuration file handling

This chapter has been superseded by ‘Configuration files’ on page 72.

Chapter 9

Shared libraries

Packages containing shared libraries must be constructed with a little care to make sure that the shared library is always available. This is especially important for packages whose shared libraries are vitally important, such as the C library (currently `libc6`).

Firstly, the package should install the shared libraries under their normal names. For example, the `libgdbmg1` package should install `libgdbm.so.1.7.3` as `/usr/lib/libgdbm.so.1.7.3`. The files should not be renamed or re-linked by any `prerm` or `postrm` scripts; `dpkg` will take care of renaming things safely without affecting running programs, and attempts to interfere with this are likely to lead to problems.

Secondly, the package should include the symbolic link that `ldconfig` would create for the shared libraries. For example, the `libgdbmg1` package should include a symbolic link from `/usr/lib/libgdbm.so.1` to `libgdbm.so.1.7.3`. This is needed so that the dynamic linker (for example `ld.so` or `ld-linux.so.*`) can find the library between the time that `dpkg` installs it and the time that `ldconfig` is run in the `postinst` script.[1]

Thirdly, the associated development package should contain a symlink for the shared library without a version number. For example, the `libgdbmg1-dev` package should include a symlink from `/usr/lib/libgdbm.so` to `libgdbm.so.1.7.3`. This symlink is needed by the linker (`ld`) when compiling packages, as it will only look for `libgdbm.so` when compiling dynamically.

Any package installing shared libraries in one of the default library directories of the dynamic linker (which are currently `/usr/lib` and `/lib`) or a directory that is listed in `/etc/ld.so.conf`[2] must call `ld-`

[1]The package management system requires the library to be placed before the symbolic link pointing to it in the `.deb` file. This is so that when `dpkg` comes to install the symlink (overwriting the previous symlink pointing at an older version of the library), the new shared library is already in place. In the past, this was achieved by creating the library in the temporary packaging directory before creating the symlink. Unfortunately, this was not always effective, since the building of the tar file in the `.deb` depended on the behavior of the underlying file system. Some file systems (such as reiserfs) reorder the files so that the order of creation is forgotten. Starting with release `1.7.0`, `dpkg` will reorder the files itself as necessary when building a package. Thus it is no longer important to concern oneself with the order of file creation.

[2]These are currently

config in its postinst script if the first argument is configure and should call it in the postrm script if the first argument is remove.

However, postrm and preinst scripts *must not* call ldconfig in the case where the package is being upgraded (see 'Details of unpack phase of installation or upgrade' on page 33 for details), as ldconfig will see the temporary names that dpkg uses for the files while it is installing them and will make the shared library links point to them, just before dpkg continues the installation and renames the temporary files!

9.1 Handling shared library dependencies - the shlibs system

If a package contains a binary or library which links to a shared library, we must ensure that when the package is installed on the system, all of the libraries needed are also installed. This requirement led to the creation of the shlibs system, which is very simple in its design: any package which *provides* a shared library also provides information on the package dependencies required to ensure the presence of this library, and any package which *uses* a shared library uses this information to determine the dependencies it requires. The files which contain the mapping from shared libraries to the necessary dependency information are called shlibs files.

Thus, when a package is built which contains any shared libraries, it must provide a shlibs file for other packages to use, and when a package is built which contains any shared libraries or compiled binaries, it must run dpkg-shlibdeps on these to determine the libraries used and hence the dependencies needed by this package.[3]

In the following sections, we will first describe where the various shlibs files are to be found, then how to use dpkg-shlibdeps, and finally the shlibs file format and how to create them if your package contains a shared library.

- /usr/X11R6/lib/Xaw3d
- /usr/local/lib
- /usr/lib/libc5-compat
- /lib/libc5-compat
- /usr/X11R6/lib

[3]In the past, the shared libraries linked to were determined by calling ldd, but now objdump to do this. The only change this makes to package building is that dpkg-shlibdeps must also be run on shared libraries, whereas in the past this was unnecessary. The rest of this footnote explains the advantage that this method gives. We say that a binary foo *directly* uses a library libbar if it is explicitly linked with that library (that is, it uses the flag -lbar during the linking stage). Other libraries that are needed by libbar are linked *indirectly* to foo, and the dynamic linker will load them automatically when it loads libbar. A package should needs to depend on the libraries it directly uses, and the dependencies for those libraries should automatically pull in the other libraries. Unfortunately, the ldd program shows both the directly and indirectly used libraries, meaning that the dependencies determined included both direct and indirect dependencies. The use of objdump avoids this problem by determining only the directly used libraries. A good example of where this helps is the following. We could update libimlib with a new version that supports a new graphics format called dgf (but retaining the same major version number). If we used the old ldd method, every package that uses libimlib would need to be recompiled so it would also depend on libdgf or it wouldn't run due to missing symbols. However with the new system, packages using libimlib can rely on libimlib itself having the dependency on libdgf and so they would not need rebuilding.

9.2 The `shlibs` files present on the system

There are several places where `shlibs` files are found. The following list gives them in the order in which they are read by `dpkg-shlibdeps`. (The first one which gives the required information is used.)

- `debian/shlibs.local`

  This lists overrides for this package. Its use is described below (see 'Writing the `debian/shlibs.local` file' on page 51).

- `/etc/dpkg/shlibs.override`

  This lists global overrides. This list is normally empty. It is maintained by the local system administrator.

- `DEBIAN/shlibs` files in the 'build directory'

  When packages are being built, any `debian/shlibs` files are copied into the control file area of the temporary build directory and given the name `shlibs`. These files give details of any shared libraries included in the package.[4]

- `/var/lib/dpkg/info/*.shlibs`

  These are the `shlibs` files corresponding to all of the packages installed on the system, and are maintained by the relevant package maintainers.

- `/etc/dpkg/shlibs.default`

  This file lists any shared libraries whose packages have failed to provide correct `shlibs` files. It was used when the `shlibs` setup was first introduced, but it is now normally empty. It is maintained by the `dpkg` maintainer.

9.3 How to use `dpkg-shlibdeps` and the `shlibs` files

Put a call to `dpkg-shlibdeps` into your `debian/rules` file. If your package contains only compiled binaries and libraries (but no scripts), you can use a command such as:

[4] An example may help here. Let us say that the source package `foo` generates two binary packages, `libfoo2` and `foo-runtime`. When building the binary packages, the two packages are created in the directories `debian/libfoo2` and `debian/foo-runtime` respectively. (`debian/tmp` could be used instead of one of these.) Since `libfoo2` provides the `libfoo` shared library, it will require a `shlibs` file, which will be installed in `debian/libfoo2/DEBIAN/shlibs`, eventually to become `/var/lib/dpkg/info/libfoo2.shlibs`. Then when `dpkg-shlibdeps` is run on the executable `debian/foo-runtime/usr/bin/foo-prog`, it will examine the `debian/libfoo2/DEBIAN/shlibs` file to determine whether `foo-prog`'s library dependencies are satisfied by any of the libraries provided by `libfoo2`. For this reason, `dpkg-shlibdeps` must only be run once all of the individual binary packages' `shlibs` files have been installed into the build directory.

```
dpkg-shlibdeps debian/tmp/usr/bin/* debian/tmp/usr/sbin/* \
  debian/tmp/usr/lib/*
```

Otherwise, you will need to explicitly list the compiled binaries and libraries.[5]

This command puts the dependency information into the `debian/substvars` file, which is then used by `dpkg-gencontrol`. You will need to place a `${shlib:Depends}` variable in the `Depends` field in the control file for this to work.

If `dpkg-shlibdeps` doesn't complain, you're done. If it does complain you might need to create your own `debian/shlibs.local` file, as explained below (see 'Writing the `debian/shlibs.local` file' on the facing page).

If you have multiple binary packages, you will need to call `dpkg-shlibdeps` on each one which contains compiled libraries or binaries. In such a case, you will need to use the `-T` option to the `dpkg` utilities to specify a different `substvars` file. For more details on this and other options, see `dpkg-shlibdeps(1)`.

9.4 The `shlibs` File Format

Each `shlibs` file has the same format. Lines beginning with # are considered to be commments and are ignored. Each line is of the form:

```
library-name soname-version-number dependencies ...
```

We will explain this by reference to the example of the `zlib1g` package, which (at the time of writing) installs the shared library `/usr/lib/libz.so.1.1.3`.

library-name is the name of the shared library, in this case `libz`. (This must match the name part of the soname, see below.)

soname-version-number is the version part of the soname of the library. The soname is the thing that must exactly match for the library to be recognized by the dynamic linker, and is usually of the form *`name`*`.so.`*`major-version`*, in our example, `libz.so.1`.[6] The version part is the part which comes after `.so.`, so in our case, it is `1`.

dependencies has the same syntax as a dependency field in a binary package control file. It should give details of which packages are required to satisfy a binary built against the version of the library contained in the package. See 'Syntax of relationship fields' on page 37 for details.

[5] If you are using `debhelper`, the `dh_shlibdeps` program will do this work for you. It will also correctly handle multi-binary packages.

[6] This can be determined using the command

```
objdump -p /usr/lib/libz.so.1.1.3 | grep SONAME
```

In our example, if the first version of the `zlib1g` package which contained a minor number of at least `1.3` was *1:1.1.3-1*, then the `shlibs` entry for this library could say:

```
libz 1 zlib1g (>= 1:1.1.3)
```

The version-specific dependency is to avoid warnings from the dynamic linker about using older shared libraries with newer binaries.

9.5 Providing a `shlibs` file

If your package provides a shared library, you should create a `shlibs` file following the format described above. It is usual to call this file `debian/shlibs` (but if you have multiple binary packages, you might want to call it `debian/shlibs.`*`package`* instead). Then let `debian/rules` install it in the control area:

```
install -m644 debian/shlibs debian/tmp/DEBIAN
```

or, in the case of a multi-binary package:

```
install -m644 debian/shlibs.package debian/package/DEBIAN/shlibs
```

An alternative way of doing this is to create the `shlibs` file in the control area directly from `debian/rules` without using a `debian/shlibs` file at all,[7] since the `debian/shlibs` file itself is ignored by `dpkg-shlibdeps`.

As `dpkg-shlibdeps` reads the `DEBIAN/shlibs` files in all of the binary packages being built from this source package, all of the `DEBIAN/shlibs` files should be installed before `dpkg-shlibdeps` is called on any of the binary packages.

9.6 Writing the `debian/shlibs.local` file

This file is intended only as a *temporary* fix if your binaries or libraries depend on a library whose package does not yet provide a correct `shlibs` file.

We will assume that you are trying to package a binary `foo`. When you try running `dpkg-shlibdeps` you get the following error message (`-O` displays the dependency information on `stdout` instead of writing it to `debian/substvars`, and the lines have been wrapped for ease of reading):

```
$ dpkg-shlibdeps -O debian/tmp/usr/bin/foo
dpkg-shlibdeps: warning: unable to find dependency
  information for shared library libbar (soname 1,
  path /usr/lib/libbar.so.1, dependency field Depends)
shlibs:Depends=libc6 (>= 2.2.2-2)
```

[7] This is what `dh_makeshlibs` in the `debhelper` suite does.

You can then run `ldd` on the binary to find the full location of the library concerned:

```
$ ldd foo
libbar.so.1 => /usr/lib/libbar.so.1 (0x4001e000)
libc.so.6 => /lib/libc.so.6 (0x40032000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

So the `foo` binary depends on the `libbar` shared library, but no package seems to provide a `*.shlibs` file handling `libbar.so.1` in `/var/lib/dpkg/info/`. Let's determine the package responsible:

```
$ dpkg -S /usr/lib/libbar.so.1
bar1: /usr/lib/libbar.so.1
$ dpkg -s bar1 | grep Version
Version: 1.0-1
```

This tells us that the `bar1` package, version 1.0-1, is the one we are using. Now we can file a bug against the `bar1` package and create our own `debian/shlibs.local` to locally fix the problem. Including the following line into your `debian/shlibs.local` file:

```
libbar 1 bar1 (>= 1.0-1)
```

should allow the package build to work.

As soon as the maintainer of `bar1` provides a correct `shlibs` file, you should remove this line from your `debian/shlibs.local` file. (You should probably also then have a versioned `Build-Depends` on `bar1` to help ensure that others do not have the same problem building your package.)

Chapter 10

The Operating System

10.1 Filesystem hierarchy

10.1.1 Filesystem Structure

The location of all installed files and directories must comply with the Filesystem Hierarchy Standard (FHS), except where doing so would violate other terms of Debian Policy. The latest version of this document can be found in the `debian-policy` package or on FHS (Debian copy) (`http://www.debian.org/doc/packaging-manuals/fhs`) alongside this manual or on FHS (upstream) (`http://www.pathname.com/fhs/`). Specific questions about following the standard may be asked on the `debian-devel` mailing list, or referred to Daniel Quinlan, the FHS coordinator, at <`quinlan@pathname.com`>.

10.1.2 Site-specific programs

As mandated by the FHS, packages must not place any files in `/usr/local`, either by putting them in the file system archive to be unpacked by `dpkg` or by manipulating them in their maintainer scripts.

However, the package may create empty directories below `/usr/local` so that the system administrator knows where to place site-specific files. These directories should be removed on package removal if they are empty.

Note, that this applies only to directories *below* `/usr/local`, not *in* `/usr/local`. Packages must not create sub-directories in the directory `/usr/local` itself, except those listed in FHS, section 4.5. However, you may create directories below them as you wish. You must not remove any of the directories listed in 4.5, even if you created them.

Since `/usr/local` can be mounted read-only from a remote server, these directories must be created and removed by the `postinst` and `prerm` maintainer scripts and not be included in the `.deb` archive. These scripts must not fail if either of these operations fail.

For example, the `emacsen-common` package could contain something like

```
if [ ! -e /usr/local/share/emacs ]
then
  if mkdir /usr/local/share/emacs 2>/dev/null
  then
    chown root:staff /usr/local/share/emacs
    chmod 2775 /usr/local/share/emacs
  fi
fi
```

in its `postinst` script, and

```
rmdir /usr/local/share/emacs/site-lisp 2>/dev/null || true
rmdir /usr/local/share/emacs 2>/dev/null || true
```

in the `prerm` script. (Note that this form is used to ensure that if the script is interrupted, the directory `/usr/local/share/emacs` will still be removed.)

If you do create a directory in `/usr/local` for local additions to a package, you should ensure that settings in `/usr/local` take precedence over the equivalents in `/usr`.

However, because `/usr/local` and its contents are for exclusive use of the local administrator, a package must not rely on the presence or absence of files or directories in `/usr/local` for normal operation.

The `/usr/local` directory itself and all the subdirectories created by the package should (by default) have permissions 2775 (group-writable and set-group-id) and be owned by `root.staff`.

10.1.3 The system-wide mail directory

The system-wide mail directory is `/var/mail`. This directory is part of the base system and should not owned by any particular mail agents. The use of the old location `/var/spool/mail` is deprecated, even though the spool may still be physically located there. To maintain partial upgrade compatibility for systems which have `/var/spool/mail` as their physical mail spool, packages using `/var/mail` must depend on either `libc6` (>= 2.1.3-13), or on `base-files` (>= 2.2.0), or on later versions of either one of these packages.

10.2 Users and groups

10.2.1 Introduction

The Debian system can be configured to use either plain or shadow passwords.

Some user ids (UIDs) and group ids (GIDs) are reserved globally for use by certain packages. Because some packages need to include files which are owned by these users or groups, or need the ids compiled into binaries, these ids must be used on any Debian system only for the purpose for which they are allocated. This is a serious restriction, and we should avoid getting in the way of local administration policies. In particular, many sites allocate users and/or local system groups starting at 100.

Apart from this we should have dynamically allocated ids, which should by default be arranged in some sensible order, but the behavior should be configurable.

Packages other than `base-passwd` must not modify `/etc/passwd`, `/etc/shadow`, `/etc/group` or `/etc/gshadow`.

10.2.2 UID and GID classes

The UID and GID numbers are divided into classes as follows:

0-99: Globally allocated by the Debian project, the same on every Debian system. These ids will appear in the `passwd` and `group` files of all Debian systems, new ids in this range being added automatically as the `base-passwd` package is updated.

Packages which need a single statically allocated uid or gid should use one of these; their maintainers should ask the `base-passwd` maintainer for ids.

100-999: Dynamically allocated system users and groups. Packages which need a user or group, but can have this user or group allocated dynamically and differently on each system, should use `adduser --system` to create the group and/or user. `adduser` will check for the existence of the user or group, and if necessary choose an unused id based on the ranges specified in `adduser.conf`.

1000-29999: Dynamically allocated user accounts. By default `adduser` will choose UIDs and GIDs for user accounts in this range, though `adduser.conf` may be used to modify this behavior.

30000-59999: Reserved.

60000-64999: Globally allocated by the Debian project, but only created on demand. The ids are allocated centrally and statically, but the actual accounts are only created on users' systems on demand.

These ids are for packages which are obscure or which require many statically-allocated ids. These packages should check for and create the accounts in `/etc/passwd` or `/etc/group` (using `adduser` if it has this facility) if necessary. Packages which are likely to require further allocations should have a 'hole' left after them in the allocation, to give them room to grow.

65000-65533: Reserved.

65534: User `nobody`. The corresponding gid refers to the group `nogroup`.

65535: `(uid_t)(-1) == (gid_t)(-1)` *must not* be used, because it is the error return sentinel value.

10.3 System run levels and `init.d` scripts

10.3.1 Introduction

The `/etc/init.d` directory contains the scripts executed by `init` at boot time and when the init state (or 'runlevel') is changed (see `init(8)`).

There are at least two different, yet functionally equivalent, ways of handling these scripts. For the sake of simplicity, this document describes only the symbolic link method. However, it must not be assumed by maintainer scripts that this method is being used, and any automated manipulation of the various runlevel behaviours by maintainer scripts must be performed using `update-rc.d` as described below and not by manually installing or removing symlinks. For information on the implementation details of the other method, implemented in the `file-rc` package, please refer to the documentation of that package.

These scripts are referenced by symbolic links in the `/etc/rc`*`n`*`.d` directories. When changing runlevels, `init` looks in the directory `/etc/rc`*`n`*`.d` for the scripts it should execute, where *n* is the runlevel that is being changed to, or `S` for the boot-up scripts.

The names of the links all have the form `S`*`mmscript`* or `K`*`mmscript`* where *mm* is a two-digit number and *script* is the name of the script (this should be the same as the name of the actual script in `/etc/init.d`).

When `init` changes runlevel first the targets of the links whose names start with a `K` are executed, each with the single argument `stop`, followed by the scripts prefixed with an `S`, each with the single argument `start`. (The links are those in the `/etc/rc`*`n`*`.d` directory corresponding to the new runlevel.) The `K` links are responsible for killing services and the `S` link for starting services upon entering the runlevel.

For example, if we are changing from runlevel 2 to runlevel 3, init will first execute all of the `K` prefixed scripts it finds in `/etc/rc3.d`, and then all of the `S` prefixed scripts in that directory. The links starting with `K` will cause the referred-to file to be executed with an argument of `stop`, and the `S` links with an argument of `start`.

The two-digit number *mm* is used to determine the order in which to run the scripts: low-numbered links have their scripts run first. For example, the `K20` scripts will be executed before the `K30` scripts. This is used when a certain service must be started before another. For example, the name server `bind` might need to be started before the news server `inn` so that `inn` can set up its access lists. In this case, the script that starts `bind` would have a lower number than the script that starts `inn` so that it runs first:

```
/etc/rc2.d/S17bind
/etc/rc2.d/S70inn
```

The two runlevels 0 (halt) and 6 (reboot) are slightly different. In these runlevels, the links with an `S` prefix are still called after those with a `K` prefix, but they too are called with the single argument `stop`.

Also, if the script name ends `.sh`, the script will be sourced in runlevel `S` rather that being run in a forked subprocess, but will be explicitly run by `sh` in all other runlevels.

10.3.2 Writing the scripts

Packages that include daemons for system services should place scripts in `/etc/init.d` to start or stop services at boot time or during a change of runlevel. These scripts should be named `/etc/init.d/`*`package`*, and they should accept one argument, saying what to do:

`start` start the service,

`stop` stop the service,

`restart` stop and restart the service,

`reload` cause the configuration of the service to be reloaded without actually stopping and restarting the service,

`force-reload` cause the configuration to be reloaded if the service supports this, otherwise restart the service.

The `start`, `stop`, `restart`, and `force-reload` options should be supported by all scripts in `/etc/init.d`, the `reload` option is optional.

The `init.d` scripts should ensure that they will behave sensibly if invoked with `start` when the service is already running, or with `stop` when it isn't, and that they don't kill unfortunately-named user processes. The best way to achieve this is usually to use `start-stop-daemon`.

If a service reloads its configuration automatically (as in the case of `cron`, for example), the `reload` option of the `init.d` script should behave as if the configuration has been reloaded successfully.

The `/etc/init.d` scripts should be treated as configuration files, either by marking them as `conffiles` or managing them correctly in the maintainer scripts (see 'Configuration files' on page 72). This is important since we want to give the local system administrator the chance to adapt the scripts to the local system, e.g., to disable a service without de-installing the package, or to specify some special command line options when starting a service, while making sure her changes aren't lost during the next package upgrade.

These scripts should not fail obscurely when the configuration files remain but the package has been removed, as configuration files remain on the system after the package has been removed. Only when `dpkg` is executed with the `--purge` option will configuration files be removed. In particular, as the `/etc/init.d/`*`package`* script itself is usually a `conffile`, it will remain on the system if the package is removed but not purged. Therefore, you should include a `test` statement at the top of the script, like this:

```
test -f program-executed-later-in-script || exit 0
```

Often there are some variables in the `init.d` scripts whose values control the bahaviour of the scripts, and which a system administrator is likely to want to change. As the scripts themselves are frequently `conffiles`, modifying them requires that the administrator merge in their changes each time the package is upgraded and the `conffile` changes. To ease the burden on the system administrator, such configurable values should not be placed directly in the script. Instead, they should be placed in a file in `/etc/default`, which typically will have the same base name as the `init.d` script. This extra file should be sourced by the script when the script runs. It must contain only variable settings and comments in POSIX `sh` format. It may either be a `conffile` or a configuration file maintained by the package maintainer scripts. See 'Configuration files' on page 72 for more details.

To ensure that vital configurable values are always available, the `init.d` script should set default values for each of the shell variables it uses, either before sourcing the `/etc/default/` file or afterwards using something like the `: ${VAR:=default}` syntax. Also, the `init.d` script must behave sensibly and not fail if the `/etc/default` file is deleted.

10.3.3 Managing the links

The program `update-rc.d` is provided for package maintainers to arrange for the proper creation and removal of `/etc/rcn.d` symbolic links, or their functional equivalent if another method is being used. This may be used by maintainers in their packages' `postinst` and `postrm` scripts.

You must not include any `/etc/rcn.d` symbolic links in the actual archive or manually create or remove the symbolic links in maintainer scripts; you must use the `update-rc.d` program instead. (The former will fail if an alternative method of maintaining runlevel information is being used.) You must not include the `/etc/rcn.d` directories themselves in the archive either. (Only the `sysvinit` package may do so.)

By default `update-rc.d` will start services in each of the multi-user state runlevels (2, 3, 4, and 5) and stop them in the halt runlevel (0), the single-user runlevel (1) and the reboot runlevel (6). The system administrator will have the opportunity to customize runlevels by simply adding, moving, or removing the symbolic links in `/etc/rcn.d` if symbolic links are being used, or by modifying `/etc/runlevel.conf` if the `file-rc` method is being used.

To get the default behavior for your package, put in your `postinst` script

```
update-rc.d package defaults >/dev/null
```

and in your `postrm`

```
if [ "$1" = purge ]; then
  update-rc.d package remove >/dev/null
fi
```

This will use a default sequence number of 20. If it does not matter when or in which order the `init.d` script is run, use this default. If it does, then you should talk to the maintainer of the `sysvinit` package or post to `debian-devel`, and they will help you choose a number.

For more information about using `update-rc.d`, please consult its manpage `update-rc.d(8)`.

10.3.4 Boot-time initialization

There used to be another directory, `/etc/rc.boot`, which contained scripts which were run once per machine boot. This has been deprecated in favour of links from `/etc/rcS.d` to files in `/etc/init.d` as described in 'Introduction' on page 56. Packages must not place files in `/etc/rc.boot`.

10.3.5 Example

The `bind` DNS (nameserver) package wants to make sure that the nameserver is running in multiuser runlevels, and is properly shut down with the system. It puts a script in `/etc/init.d`, naming the script appropriately `bind`. As you can see, the script interprets the argument `reload` to send the nameserver a `HUP` signal (causing it to reload its configuration); this way the system administrator can say `/etc/init.d/bind reload` to reload the name server. The script has one configurable value, which can be used to pass parameters to the named program at startup; this value is read from `/etc/default/bind` (see below).

```
#!/bin/sh
#
# Original version by Robert Leslie
# <rob@mars.org>, edited by iwj and cs

test -x /usr/sbin/named || exit 0

# Source defaults file.
PARAMS=''
if [ -f /etc/default/bind ]; then
  . /etc/default/bind
fi


case "$1" in
start)
  echo -n "Starting domain name service: named"
  start-stop-daemon --start --quiet --exec /usr/sbin/named \
                    -- $PARAMS
  echo "."
  ;;
stop)
  echo -n "Stopping domain name service: named"
  start-stop-daemon --stop --quiet  \
    --pidfile /var/run/named.pid --exec /usr/sbin/named
  echo "."
```

```
    ;;
  restart)
    echo -n "Restarting domain name service: named"
    start-stop-daemon --stop --quiet  \
      --pidfile /var/run/named.pid --exec /usr/sbin/named
    start-stop-daemon --start --verbose --exec /usr/sbin/named \
                      -- $PARAMS
    echo "."
    ;;
  force-reload|reload)
    echo -n "Reloading configuration of domain name service: named"
    start-stop-daemon --stop --signal 1 --quiet  \
      --pidfile /var/run/named.pid --exec /usr/sbin/named
    echo "."
    ;;
  *)
    echo "Usage: /etc/init.d/bind {start|stop|restart|reload|force-reload}" >&2
    exit 1
    ;;
  esac

  exit 0
```

Complementing the above init script is a configuration file `/etc/default/bind`, which contains configurable parameters used by the script. This would be created by the `postinst` script if it was not already present, and removed on purge by the `postrm` script.

```
# Specified parameters to pass to named. See named(8).
# You may uncomment the following line, and edit to taste.
#PARAMS="-u nobody"
```

Another example on which you can base your `/etc/init.d` scripts is found in `/etc/init.d/skeleton`.

If this package is happy with the default setup from `update-rc.d`, namely an ordering number of 20 and having named running in all runlevels, it can say in its `postinst`:

```
update-rc.d bind defaults >/dev/null
```

And in its `postrm`, to remove the links when the package is purged:

```
if [ "$1" = purge ]; then
  update-rc.d bind remove >/dev/null
fi
```

10.4 Console messages from `init.d` scripts

This section describes the formats to be used for messages written to standard output by the `/etc/init.d` scripts. The intent is to improve the consistency of Debian's startup and shutdown look and feel. For this reason, please look very carefully at the details. We want the messages to have the same format in terms of wording, spaces, punctuation and case of letters.

Here is a list of overall rules that you should use when you create output messages. They can be useful if you have a non-standard message that is not covered specifically in the sections below.

- Every message should fit in one line (fewer than 80 characters), start with a capital letter and end with a period (`.`) and line feed (`"\n"`).
- If you want to express that the computer is working on something (that is, performing a specific task, not starting or stopping a program), we use an "ellipsis" (three dots: `...`). Note that we don't insert spaces before or after the dots. If the task has been completed we write `done.` and a line feed.
- Design your messages as if the computer is telling you what he is doing (let him be polite :-), but don't mention "him" directly. For example, if you think of saying

  ```
  I'm starting network daemons: nfsd mountd.
  ```

  just say

  ```
  Starting network daemons: nfsd mountd.
  ```

There are standard message formats for the following situations. They should be used by the `init.d` scripts.

- When daemons are started

  If your script starts one or more daemons, the output should look like this (a single line, no leading spaces):

  `Starting` *`description`*`:` *`daemon-1`* `...` *`daemon-n`*`.`

  The *description* should describe the subsystem the daemon or set of daemons are part of, while *daemon-1* up to *daemon-n* denote each daemon's name (typically the file name of the program).

  For example, the output of `/etc/init.d/lpd` would look like:

  ```
  Starting printer spooler: lpd.
  ```

  This can be achieved by saying

```
echo -n "Starting printer spooler: lpd"
start-stop-daemon --start --quiet --exec /usr/sbin/lpd
echo "."
```

in the script. If you have more than one daemon to start, you should do the following:

```
echo -n "Starting remote file system services:"
echo -n " nfsd"; start-stop-daemon --start --quiet nfsd
echo -n " mountd"; start-stop-daemon --start --quiet mountd
echo -n " ugidd"; start-stop-daemon --start --quiet ugidd
echo "."
```

This makes it possible for the user to see what takes so long and when the final daemon has been started. You should be careful where to put spaces: in the example above the system administrator can easily comment out a line if he don't wants to start a specific daemon, while the displayed message still looks good.

- When a system parameter is being set

  If you have to set up different system parameters during the system boot, you should use this format:

  `Setting` *`parameter`* `to` `` ` ``*`value`*`'.`

  You can use a statement such as the following to get the quotes right:

  ```
  echo "Setting DNS domainname to \`$domainname'."
  ```

  Note that the left quotation mark (`) is different from the right one (').

- When a daemon is stopped or restarted

  When you stop or restart a daemon, you should issue a message identical to the startup message, except that `Starting` is replaced with `Stopping` or `Restarting` respectively.

  For example, stopping the printer daemon will like like this:

  ```
  Stopping printer spooler: lpd.
  ```

- When something is executed

  There are several examples where you have to run a program at system startup or shutdown to perform a specific task, for example, setting the system's clock using `netdate` or killing all processes when the system shuts down. Your message should look like this:

  ```
  Doing something very useful...done.
  ```

  You should print the `done.` immediately after the job has been completed, so that the user is informed why she has to wait. You can get this behavior by saying

```
echo -n "Doing something very useful..."
do_something
echo "done."
```

in your script.

- When the configuration is reloaded

  When a daemon is forced to reload its configuration files you should use the following format:

  ```
  Reloading description configuration...done.
  ```

  where *description* is the same as in the daemon starting message.

10.5 Cron jobs

Packages must not modify the configuration file `/etc/crontab`, and they must not modify the files in `/var/spool/cron/crontabs`.

If a package wants to install a job that has to be executed via cron, it should place a file with the name of the package in one or more of the following directories:

```
/etc/cron.daily
/etc/cron.weekly
/etc/cron.monthly
```

As these directory names imply, the files within them are executed on a daily, weekly, or monthly basis, respectively. The exact times are listed in `/etc/crontab`.

All files installed in any of these directories must be scripts (e.g., shell scripts or Perl scripts) so that they can easily be modified by the local system administrator. In addition, they should be treated as configuration files.

If a certain job has to be executed more frequently than daily, the package should install a file `/etc/cron.d/`*`package`*. This file uses the same syntax as `/etc/crontab` and is processed by `cron` automatically. The file must also be treated as a configuration file. (Note that entries in the `/etc/cron.d` directory are not handled by `anacron`. Thus, you should only use this directory for jobs which may be skipped if the system is not running.)

The scripts or crontab entries in these directories should check if all necessary programs are installed before they try to execute them. Otherwise, problems will arise when a package was removed but not purged since configuration files are kept on the system in this situation.

10.6 Menus

Menu entries should follow the current menu policy found in the `menu-policy` files in the `debian-policy` package. It may also be found on the Debian FTP site `ftp.debian.org` as the file `/debian/doc/package-developer/menu-policy.txt.gz`, or in the equivalent location on your local mirror.

The Debian `menu` package provides a standard interface between packages providing applications and documents, and *menu programs* (either X window managers or text-based menu programs such as `pdmenu`).

All packages that provide applications that need not be passed any special command line arguments for normal operation should register a menu entry for those applications, so that users of the `menu` package will automatically get menu entries in their window managers, as well in shells like `pdmenu`.

Please also refer to the *Debian Menu System* documentation that comes with the `menu` package for information about how to register your applications and web documents.

10.7 Multimedia handlers

Packages which provide the ability to view/show/play, compose, edit or print MIME types should register themselves as such following the current MIME support policy found in the `mime-policy` files in the `debian-policy` package. It may also be found on the Debian FTP site `ftp.debian.org` as the file `/debian/doc/package-developer/mime-policy.txt.gz`, or in the equivalent location on your local mirror.

MIME (Multipurpose Internet Mail Extensions, RFCs 2045-2049) is a mechanism for encoding files and data streams and providing meta-information about them, in particular their type (e.g. audio or video) and format (e.g. PNG, HTML, MP3).

Registration of MIME type handlers allows programs like mail user agents and web browsers to to invoke these handlers to view, edit or display MIME types they don't support directly.

10.8 Keyboard configuration

To achieve a consistent keyboard configuration so that all applications interpret a keyboard event the same way, all programs in the Debian distribution must be configured to comply with the following guidelines.

The following keys must have the specified interpretations:

<-- delete the character to the left of the cursor

Delete delete the character to the right of the cursor

`Control+H` emacs: the help prefix

The interpretation of any keyboard events should be independent of the terminal that is used, be it a virtual console, an X terminal emulator, an rlogin/telnet session, etc.

The following list explains how the different programs should be set up to achieve this:

- <-- generates `KB_Backspace` in X.
- `Delete` generates `KB_Delete` in X.
- X translations are set up to make `KB_Backspace` generate ASCII DEL, and to make `KB_Delete` generate `ESC [ 3 ~` (this is the vt220 escape code for the ‘delete character’ key). This must be done by loading the X resources using `xrdb` on all local X displays, not using the application defaults, so that the translation resources used correspond to the `xmodmap` settings.
- The Linux console is configured to make <-- generate DEL, and `Delete` generate `ESC [ 3 ~`.
- X applications are configured so that < deletes left, and `Delete` deletes right. Motif applications already work like this.
- Terminals should have `stty erase ^?` .
- The `xterm` terminfo entry should have `ESC [ 3 ~` for `kdch1`, just as for `TERM=linux` and `TERM=vt220`.
- Emacs is programmed to map `KB_Backspace` or the `stty erase` character to `delete-backward-char`, and `KB_Delete` or `kdch1` to `delete-forward-char`, and `^H` to `help` as always.
- Other applications use the `stty erase` character and `kdch1` for the two delete keys, with ASCII DEL being ‘delete previous character’ and `kdch1` being ‘delete character under cursor’.

This will solve the problem except for the following cases:

- Some terminals have a <-- key that cannot be made to produce anything except `^H`. On these terminals Emacs help will be unavailable on `^H` (assuming that the `stty erase` character takes precedence in Emacs, and has been set correctly). `M-x help` or `F1` (if available) can be used instead.
- Some operating systems use `^H` for `stty erase`. However, modern telnet versions and all rlogin versions propagate `stty` settings, and almost all UNIX versions honour `stty erase`. Where the `stty` settings are not propagated correctly, things can be made to work by using `stty` manually.
- Some systems (including previous Debian versions) use `xmodmap` to arrange for both <-- and `Delete` to generate `KB_Delete`. We can change the behavior of their X clients using the same X resources that we use to do it for our own clients, or configure our clients using their resources when things are the other way around. On displays configured like this `Delete` will not work, but <-- will.

- Some operating systems have different `kdch1` settings in their `terminfo` database for `xterm` and others. On these systems the `Delete` key will not work correctly when you log in from a system conforming to our policy, but `<--` will.

10.9 Environment variables

A program must not depend on environment variables to get reasonable defaults. (That's because these environment variables would have to be set in a system-wide configuration file like `/etc/profile`, which is not supported by all shells.)

If a program usually depends on environment variables for its configuration, the program should be changed to fall back to a reasonable default configuration if these environment variables are not present. If this cannot be done easily (e.g., if the source code of a non-free program is not available), the program must be replaced by a small 'wrapper' shell script which sets the environment variables if they are not already defined, and calls the original program.

Here is an example of a wrapper script for this purpose:

```
#!/bin/sh
BAR=${BAR:-/var/lib/fubar}
export BAR
exec /usr/lib/foo/foo "$@"
```

Furthermore, as `/etc/profile` is a configuration file of the `base-files` package, other packages must not put any environment variables or other commands into that file.

Chapter 11

Files

11.1 Binaries

Two different packages must not install programs with different functionality but with the same filenames. (The case of two programs having the same functionality but different implementations is handled via 'alternatives' or the 'Conflicts' mechanism. See 'Maintainer scripts' on page 10 and 'Conflicting binary packages - `Conflicts`' on page 40 respectively.) If this case happens, one of the programs must be renamed. The maintainers should report this to the `debian-devel` mailing list and try to find a consensus about which program will have to be renamed. If a consensus cannot be reached, *both* programs must be renamed.

Generally the following compilation parameters should be used:

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

Note that by default all installed binaries should be stripped, 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.

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.

Debugging symbols are useful for error diagnosis, investigation of core dumps (which may be submitted by users in bug reports), or testing and developing the software. Therefore it is recommended to support building the package with debugging information through the following interface: If the environment variable `DEB_BUILD_OPTIONS` contains the string `debug`, compile the software with debugging information (usually this involves adding the `-g` flag to `CFLAGS`). This allows the generation of a build tree with debugging information. If the environment variable `DEB_BUILD_OPTIONS` contains the string `nostrip`, do not strip the files at installation time. This allows one to generate a package with debugging information included.[1] The following makefile snippet is an example of how one may test for either condition; you will probably have to massage this example in order to make it work for your package.

```
CFLAGS = -O2 -Wall
INSTALL = install
INSTALL_FILE    = $(INSTALL) -p    -o root -g root  -m  644
INSTALL_PROGRAM = $(INSTALL) -p    -o root -g root  -m  755
INSTALL_SCRIPT  = $(INSTALL) -p    -o root -g root  -m  755
INSTALL_DIR     = $(INSTALL) -p -d -o root -g root  -m  755

ifneq (,$(findstring debug,$(DEB_BUILD_OPTIONS)))
CFLAGS += -g
endif
ifeq (,$(findstring nostrip,$(DEB_BUILD_OPTIONS)))
INSTALL_PROGRAM += -s
endif
```

It is up to the package maintainer to decide what compilation options are best for the package. Certain binaries (such as computationally-intensive programs) will 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.

11.2 Libraries

All libraries must have a shared version in the `lib*` package and a static version in the `lib*-dev` package. The shared version must be compiled with `-fPIC`, and the static version must not be. In other words, each `*.c` file will need to be compiled twice.

You must specify the gcc option `-D_REENTRANT` when building a library (either static or shared) to make the library compatible with LinuxThreads.

Note that all installed shared libraries should be stripped with

```
strip --strip-unneeded your-lib
```

(The option `--strip-unneeded` makes `strip` remove only the symbols which aren't needed for relocation processing.) Shared libraries can function perfectly well when stripped, since the symbols for

[1] Rationale: Using `-g` by default causes wasted CPU cycles since the information is stripped away anyway; this can have a significant impact on the efficiency of the autobuilders. Having a standard way to build a debugging variant also makes it easier to build debugging bins and libraries since it provides a documented way of getting this type of build; one does not have to manually edit `debian/rules` or `Makefiles`.

dynamic linking are in a separate part of the ELF object file.[2]

Note that under some circumstances it may be useful to install a shared library unstripped, for example when building a separate package to support debugging.

Shared object files (often `.so` files) that are not public libraries, that is, they are not meant to be linked to by third party executables (binaries of other packages), should be installed in subdirectories of the `/usr/lib` directory. Such files are exempt from the rules that govern ordinary shared libraries, except that they must not be installed executable and should be stripped.[3]

Packages containing shared libraries that may be linked to by other packages' binaries, but which for some *compelling* reason can not be installed in `/usr/lib` directory, may install the shared library files in subdirectories of the `/usr/lib` directory, in which case they should arrange to add that directory in `/etc/ld.so.conf` in the package's post-installation script, and remove it in the package's post-removal script.

An ever increasing number of packages are using `libtool` to do their linking. The latest GNU libtools (>= 1.3a) can take advantage of the metadata in the installed `libtool` archive files (`*.la` files). The main advantage of `libtool`'s `.la` files is that it allows `libtool` to store and subsequently access metadata with respect to the libraries it builds. `libtool` will search for those files, which contain a lot of useful information about a library (such as library dependency information for static linking). Also, they're *essential* for programs using `libltdl`.[4]

Packages that use `libtool` to create shared libraries should include the `.la` files in the `-dev` package, unless the package relies on `libtool`'s `libltdl` library, in which case the `.la` files must go in the run-time library package.

You must 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.

11.3 Shared libraries

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

[2] You might also want to use the options `--remove-section=.comment` and `--remove-section=.note` on both shared libraries and executables, and `--strip-debug` on static libraries.

[3] A common example are the so-called "plug-ins", internal shared objects that are dynamically loaded by programs using `dlopen(3)`.

[4] Although `libtool` is fully capable of linking against shared libraries which don't have `.la` files, as it is a mere shell script it can add considerably to the build time of a `libtool`-using package if that shell script has to derive all this information from first principles for each library every time it is linked. With the advent of `libtool` version 1.4 (and to a lesser extent `libtool` version 1.3), the `.la` files also store information about inter-library dependencies which cannot necessarily be derived after the `.la` file is deleted.

libraries you need to create two packages: `librarynamesoversion`, where `soversion` is the version number in the soname of the shared library[5] and `librarynamesoversion-dev`.

If you prefer only to support one development version at a time you may name the development package `libraryname-dev`; otherwise you may need to use `dpkg`'s Conflicts mechanism (see 'Conflicting binary packages - `Conflicts`' on page 40) to ensure that the user only installs one development version at a time (as different development versions are likely to have the same header files in them, which would cause a filename clash if both were installed). Typically the development version should also have an exact version dependency on the runtime library, to make sure that compilation and linking happens correctly. The `${Source-Version}` substitution variable can be useful for this purpose.

Packages which use the shared library should have a dependency on the name of the shared library package, `librarynamesoversion`. When the soname changes you can have both versions of the library installed while migrating 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 *soversion* in the package name), or if the development package is small you may include them in there.

If you have several shared libraries built from the same source tree you may lump them all together into a single shared library package, provided that you change all of their sonames at once (so that you don't get filename clashes if you try to install different versions of the combined shared libraries package).

Shared libraries should not be installed executable, since the dynamic linker does not require this and trying to execute a shared library usually results in a core dump.

11.4 Scripts

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 should use `set -e` or check the exit status of *every* command.

The standard shell interpreter `/bin/sh` can be a symbolic link to any POSIX compatible shell, if `echo -n` does not generate a newline.[6] Thus, shell scripts specifying `/bin/sh` as interpreter should only use

[5] The soname is the shared object name: 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. For example, if the soname of the library is `libfoo.so.6`, the library package would be called `libfoo6`.

[6] Debian policy specifies POSIX behavior for `/bin/sh`, but `echo -n` has widespread use in the Linux community (in particular including this policy, the Linux kernel source, many Debian scripts, etc.). This `echo -n` mechanism is valid but not required under POSIX, hence this explicit addition. Also, rumour has it that this shall be mandated under the LSB anyway.

POSIX features. If a script requires non-POSIX features from the shell interpreter, the appropriate shell must be specified in the first line of the script (e.g., `#!/bin/bash`) and the package must depend on the package providing the shell (unless the shell package is marked 'Essential', as in the case of `bash`).

You may wish to restrict your script to POSIX features when possible so that it may use `/bin/sh` as its interpreter. If your script works with `ash`, it's probably POSIX compliant, but if you are in doubt, use `/bin/bash`.

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, which can be found at `http://language.perl.com/versus/csh.whynot`.[7] 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 the `c-shell` virtual package.

Any scripts which create files in world-writeable directories (e.g., in `/tmp`) must use a mechanism which will fail if a file with the same name already exists.

The Debian base system provides the `tempfile` and `mktemp` utilities for use by scripts for this purpose.

11.5 Symbolic links

In general, symbolic links within a top-level directory should be relative, and symbolic links pointing from one top-level directory into another should be absolute. (A top-level directory is a sub-directory of the root directory `/`.)

In addition, symbolic links should be specified as short as possible, i.e., link targets like `foo/../bar` are deprecated.

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`, you can 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
```

[7] It can also be found on `http://www.cpan.org/doc/FMTEYEWTK/versus/csh.whynot` or on the ftp site `ftp.cpan.org` as `/pub/perl/CPAN/doc/FMTEYEWTK/versus/csh.whynot`.

A symbolic link pointing to a compressed file should always have the same file extension as the referenced file. (For example, if a file `foo.gz` is referenced by a symbolic link, the filename of the link has to end with '`.gz`' too, as in `bar.gz`.)

11.6 Device files

Packages must not include device files in the package file tree.

If a package needs any special device files that are not included in the base system, it must call `MAKEDEV` in the `postinst` script, after asking the user for permission to do so.

Packages must not remove any device files in the `postrm` or any other script. This is left to the system administrator.

Debian uses the serial devices `/dev/ttyS*`. Programs using the old `/dev/cu*` devices should be changed to use `/dev/ttyS*`.

11.7 Configuration files

11.7.1 Definitions

configuration file A file that affects the operation of a program, or provides site- or host-specific information, or otherwise customizes the behavior of a program. Typically, configuration files are intended to be modified by the system administrator (if needed or desired) to conform to local policy or to provide more useful site-specific behavior.

`conffile` A file listed in a package's `conffiles` file, and is treated specially by `dpkg` (see 'Details of configuration' on page 36).

The distinction between these two is important; they are not interchangeable concepts. Almost all `conffiles` are configuration files, but many configuration files are not `conffiles`.

Note that a script that embeds configuration information (such as most of the files in `/etc/default` and `/etc/cron.{daily,weekly,monthly}`) is de-facto a configuration file and should be treated as such.

11.7.2 Location

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

If your package creates or uses configuration files outside of `/etc`, and it is not feasible to modify the package to use the `/etc`, you should still put the files in `/etc` and create symbolic links to those files from the location that the package requires.

11.7.3 Behavior

Configuration file handling must conform to the following behavior:

- local changes must be preserved during a package upgrade, and
- configuration files must be preserved when the package is removed, and only deleted when the package is purged.

The easy way to achieve this behavior is to make the configuration file a `conffile`. This is appropriate only if it is possible to distribute a default version that will work for most installations, although some system administrators may choose to modify it. This implies that the default version will be part of the package distribution, and must not be modified by the maintainer scripts during installation (or at any other time).

In order to ensure that local changes are preserved correctly, no package may contain or make hard links to conffiles.[8]

The other way to do it is via the maintainer scripts. In this case, the configuration file must not be listed as a `conffile` and must not be part of the package distribution. If the existence of a file is required for the package to be sensibly configured it is the responsibility of the package maintainer to provide maintainer scripts which correctly create, update and maintain the file and remove it on purge. (See 'Package maintainer scripts and installation procedure' on page 31 for more information.) These scripts must be idempotent (i.e., must work correctly if `dpkg` needs to re-run them due to errors during installation or removal), must cope with all the variety of ways `dpkg` can call maintainer scripts, must not overwrite or otherwise mangle the user's configuration without asking, must not ask unnecessary questions (particularly during upgrades), and otherwise be good citizens.

The scripts are not required to configure every possible option for the package, but only those necessary to get the package running on a given system. Ideally the sysadmin should not have to do any configuration other than that done (semi-)automatically by the `postinst` script.

A common practice is to create a script called *package*`-configure` and have the package's `postinst` call it if and only if the configuration file does not already exist. In certain cases it is useful for there to be an example or template file which the maintainer scripts use. Such files should be in `/usr/share/`*package* or `/usr/lib/`*package* (depending on whether they are architecture-independent or not). There should be symbolic links to them from `/usr/share/doc/`*package*`/examples` if they are examples, and should be perfectly ordinary `dpkg`-handled files (*not* configuration files).

These two styles of configuration file handling must not be mixed, for that way lies madness: `dpkg` will ask about overwriting the file every time the package is upgraded.

[8] Rationale: There are two problems with hard links. The first is that some editors break the link while editing one of the files, so that the two files may unwittingly become unlinked and different. The second is that `dpkg` might break the hard link while upgrading `conffiles`.

11.7.4 Sharing configuration files

Packages which specify the same file as a `conffile` must be tagged as *conflicting* with each other. (This is an instance of the general rule about not sharing files. Note that neither alternatives nor diversions are likely to be appropriate in this case; in particular, `dpkg` does not handle diverted `conffiles` well.)

The maintainer scripts must not alter a `conffile` of *any* package, including the one the scripts belong to.

If two or more packages use the same configuration file and it is reasonable for both to be installed at the same time, one of these packages must be defined as *owner* of the configuration file, i.e., it will be the package which handles that file as a configuration file. Other packages that use the configuration file must depend on the owning package if they require the configuration file to operate. If the other package will use the configuration file if present, but is capable of operating without it, no dependency need be declared.

If it is desirable for two or more related packages to share a configuration file *and* for all of the related packages to be able to modify that configuration file, then the following should be done:

1. One of the related packages (the “owning” package) will manage the configuration file with maintainer scripts as described in the previous section.
2. The owning package should also provide a program that the other packages may use to modify the configuration file.
3. The related packages must use the provided program to make any desired modifications to the configuration file. They should either depend on the core package to guarantee that the configuration modifier program is available or accept gracefully that they cannot modify the configuration file if it is not. (This is in addition to the fact that the configuration file may not even be present in the latter scenario.)

Sometimes it’s appropriate to create a new package which provides the basic infrastructure for the other packages and which manages the shared configuration files. (The `sgml-base` package is a good example.)

11.7.5 User configuration files (“dotfiles”)

The files in `/etc/skel` will automatically be copied into new user accounts by `adduser`. No other program should reference the files in `/etc/skel`.

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 treated as a configuration file.

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

Therefore, if a program in a Debian package needs to be configured in some way in order to operate sensibly, that should be done using a site-wide configuration file placed 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 may 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 (or necessarily desirable) mechanism for ensuring that the appropriate dotfiles are copied into the accounts of existing users when a package is installed.

11.8 Log files

Log files should usually be named /var/log/*package*.log. If you have many log files, or need a separate directory for permission reasons (/var/log is writable only by root), you should usually create a directory named /var/log/*package* and place your log files there.

Log files must be rotated occasionally so that they don't grow indefinitely; the best way to do this is to drop a log rotation configuration file into the directory /etc/logrotate.d and use the facilities provided by logrotate.[9] Here is a good example for a logrotate config file (for more information see logrotate(8)):

```
/var/log/foo/* {
rotate 12
weekly
compress
postrotate
/etc/init.d/foo force-reload
endscript
}
```

This rotates all files under /var/log/foo, saves 12 compressed generations, and forces the daemon to reload its configuration information after the log rotation.

Log files should be removed when the package is purged (but not when it is only removed). This should be done by the postrm script when it is called with the argument purge (see 'Details of removal and/or configuration purging' on page 36).

11.9 Permissions and owners

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 should try to be as consistent as possible with the rest of the system. You should probably also discuss it on debian-devel first.

[9]The traditional approach to log files has been to set up *ad hoc* log rotation schemes using simple shell scripts and cron. While this approach is highly customizable, it requires quite a lot of sysadmin work. Even though the original Debian system helped a little by automatically installing a system which can be used as a template, this was deemed not enough. The use of logrotate, a program developed by Red Hat, is better, as it centralizes log management. It has both a configuration file (/etc/logrotate.conf) and a directory where packages can drop their individual log rotation configurations (/etc/logrotate.d).

Files should be owned by `root.root`, and made writable only by the owner and universally readable (and executable, if appropriate), that is mode 644 or 755.

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; again there is no point in making them unreadable to those users who must not be allowed to execute them.

It is possible to arrange that the system administrator can reconfigure the package to correspond to their local security policy by changing the permissions on a binary: they can do this by using `dpkg-statoverride`, as described below.[10] Another method you should consider is to create a group for people allowed to use the program(s) and make any setuid executables executable only by that group.

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, as in this case you need a statically allocated id).

If you need a statically allocated id, you must ask for a user or group id from the `base-passwd` maintainer, and must not release the package until you have been allocated one. Once you have been allocated one you must either make the package depend on a version of the `base-passwd` package with the id present in `/etc/passwd` or `/etc/group`, or arrange for your package to create the user or group itself with the correct id (using `adduser`) in its `preinst` or `postinst`. (Doing it in the `postinst` is to be preferred if it is possible, otherwise a pre-dependency will be needed on the `adduser` package.)

On the other hand, the program might be able to determine the uid or gid from the user or group name at runtime, so that a dynamically allocated id can be used. In this case you should 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 `preinst` or `postinst` script (again, the latter is to be preferred if it is possible).

[10] Ordinary files installed by `dpkg` (as opposed to `conffiles` and other similar objects) normally have their permissions reset to the distributed permissions when the package is reinstalled. However, the use of `dpkg-statoverride` overrides this default behaviour. If you use this method, you should remember to describe `dpkg-statoverride` in the package documentation; being a relatively new addition to Debian, it is probably not yet well-known.

Note that changing the numeric value of an id associated with a name is very difficult, and involves searching the file system 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.

11.9.1 The use of `dpkg-statoverride`

This section is not intended as policy, but as a description of the use of `dpkg-statoverride`.

`dpkg-statoverride` is a replacement for the deprecated `suidmanager` package. Packages which previously used `suidmanager` should have a `Conflicts: suidmanager (« 0.50)` entry (or even `(« 0.52)`), and calls to `suidregister` and `suidunregister` should now be simply removed from the maintainer scripts.

If a system administrator wishes to have a file (or directory or other such thing) installed with owner and permissions different from those in the distributed Debian package, he can use the `dpkg-statoverride` program to instruct `dpkg` to use the different settings every time the file is installed. Thus the package maintainer should distribute the files with their normal permissions, and leave it for the system administrator to make any desired changes. For example, a daemon which is normally required to be setuid root, but in certain situations could be used without being setuid, should be installed setuid in the `.deb`. Then the local system administrator can change this if they wish. If there are two standard ways of doing it, the package maintainer can use `debconf` to find out the preference, and call `dpkg-statoverride` in the maintainer script if necessary to accommodate the system administrator's choice.

Given the above, `dpkg-statoverride` is essentially a tool for system administrators and would not normally be needed in the maintainer scripts. There is one type of situation, though, where calls to `dpkg-statoverride` would be needed in the maintainer scripts, and that involves packages which use dynamically allocated user or group ids. In such a situation, something like the following idiom can be very helpful in the package's `postinst`, where `sysuser` is a dynamically allocated id:

```
for i in /usr/bin/foo /usr/sbin/bar
do
  if ! dpkg-statoverride --list $i >/dev/null
  then
    dpkg-statoverride --update --add sysuser root 4755 $i
  fi
done
```

The corresponding `dpkg-statoverride --remove` calls can then be made unconditionally when the package is purged.

Chapter 12

Customized programs

12.1 Architecture specification strings

If a program needs to specify an *architecture specification string* in some place, the following format should be used: *arch-os*[1].

Note that we don't want to use *arch*-debian-linux to apply to the rule *architecture-vendor-os* since this would make our programs incompatible with other Linux distributions. We also don't use something like *arch*-unknown-linux, since the unknown does not look very good.

12.2 Daemons

The configuration files /etc/services, /etc/protocols, and /etc/rpc are managed by the netbase package and must not be modified by other packages.

If a package requires a new entry in one of these files, the maintainer should get in contact with the netbase maintainer, who will add the entries and release a new version of the netbase package.

The configuration file /etc/inetd.conf must not be modified by the package's scripts except via the update-inetd script or the DebianNet.pm Perl module. See their documentation for details on how to add entries.

If a package wants to install an example entry into /etc/inetd.conf, the entry must be preceded with exactly one hash character (#). Such lines are treated as 'commented out by user' by the update-inetd script and are not changed or activated during package updates.

[1] The following architectures and operating systems are currently recognised by dpkg-archictecture. The architecture, *arch*, is one of the following: alpha, arm, hppa, i386, ia64, m68k, mips, mipsel, powerpc, s390, sh, sheb, sparc and sparc64. The operating system, *os*, is one of: linux, gnu, freebsd and openbsd. Use of gnu in this string is reserved for the GNU/Hurd operating system.

12.3 Using pseudo-ttys and modifying wtmp, utmp and lastlog

Some programs need to create pseudo-ttys. This should be done using Unix98 ptys if the C library supports it. The resulting program must not be installed setuid root, unless that is required for other functionality.

The files `/var/run/utmp`, `/var/log/wtmp` and `/var/log/lastlog` must be installed writeable by group `utmp`. Programs which need to modify those files must be installed setgid `utmp`.

12.4 Editors and pagers

Some programs have the ability to launch an editor or pager program to edit or display a text document. Since there are lots of different editors and pagers available in the Debian distribution, the system administrator and each user should have the possibility to choose his/her preferred editor and pager.

In addition, every program should choose a good default editor/pager if none is selected by the user or system administrator.

Thus, every program that launches an editor or pager must use the EDITOR or PAGER environment variable to determine the editor or pager the user wishes to use. If these variables are not set, the programs `/usr/bin/editor` and `/usr/bin/pager` should be used, respectively.

These two files are managed through the `dpkg` 'alternatives' mechanism. Thus every package providing an editor or pager must call the `update-alternatives` script to register these programs.

If it is very hard to adapt a program to make use of the EDITOR or PAGER variables, that program may be configured to use `/usr/bin/sensible-editor` and `/usr/bin/sensible-pager` as the editor or pager program respectively. These are two scripts provided in the Debian base system that check the EDITOR and PAGER variables and launch the appropriate program, and fall back to `/usr/bin/editor` and `/usr/bin/pager` if the variable is not set.

A program may also use the VISUAL environment variable to determine the user's choice of editor. If it exists, it should take precedence over EDITOR. This is in fact what `/usr/bin/sensible-editor` does.

It is not required for a package to depend on `editor` and `pager`, nor is it required for a package to provide such virtual packages.[2]

12.5 Web servers and applications

This section describes the locations and URLs that should be used by all web servers and web applications in the Debian system.

[2]The Debian base system already provides an editor and a pager program,

1. Cgi-bin executable files are installed in the directory

   `/usr/lib/cgi-bin/`*`cgi-bin-name`*

   and should be referred to as

   `http://localhost/cgi-bin/`*`cgi-bin-name`*

2. Access to HTML documents

   HTML documents for a package are stored in `/usr/share/doc/`*`package`* and can be referred to as

   `http://localhost/doc/`*`package`*`/`*`filename`*

   The web server should restrict access to the document tree so that only clients on the same host can read the documents. If the web server does not support such access controls, then it should not provide access at all, or ask about providing access during installation.

3. Web Document Root

   Web Applications should try to avoid storing files in the Web Document Root. Instead they should use the /usr/share/doc/*package* directory for documents and register the Web Application via the menu package. If access to the web document root is unavoidable then use

   `/var/www`

   as the Document Root. This might be just a symbolic link to the location where the system administrator has put the real document root.

12.6 Mail transport, delivery and user agents

Debian packages which process electronic mail, whether mail user agents (MUAs) or mail transport agents (MTAs), must ensure 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/mail` and the interface to send a mail message is `/usr/sbin/sendmail` (as per the FHS). On older systems, the mail spool may be physically located in `/var/spool/mail`, but all access to the mail spool should be via the `/var/mail` symlink. The mail spool is part of the base system and not part of the MTA package.

All Debian MUAs, MTAs, MDAs and other mailbox accessing programs (such as IMAP daemons) must lock the mailbox in an NFS-safe way. This means that `fcntl()` locking must be combined with dot locking. To avoid deadlocks, a program should use `fcntl()` first and dot locking after this, or alternatively implement the two locking methods in a non blocking way[3]. Using the functions `maillock` and `mailunlock` provided by the `liblockfile*`[4] packages is the recommended way to realize this.

Mailboxes are generally mode 660 *`user`*`.mail` unless the system administrator 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 writable by group mail.

The mail spool is 2775 `root.mail`, and MUAs should be setgid mail to do the locking mentioned above (and must obviously 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 must come with a `newaliases` program, even if it does nothing, but older MTA packages did not do this so programs should not fail if `newaliases` cannot be found. Note that because of this, all MTA packages must have `Provides`, `Conflicts` and `Replaces: mail-transport-agent` control file fields.

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

The `rmail` program used by UUCP for incoming mail should be `/usr/sbin/rmail`. Likewise, `rsmtp`, for receiving batch-SMTP-over-UUCP, should be `/usr/sbin/rsmtp` if it is supported.

If your package needs to know what hostname to use on (for example) 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).

Such package should check for the existence of this file when it is being configured. If it exists, it should be used without comment, although an MTA's configuration script may wish to prompt the user even if it finds that this file exists. If the file does not exist, the package should prompt the user for the value (preferably using `debconf`) 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. For example, in this situation the `inn` package could say something like:

```
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`.

[3]If it is not possible to establish both locks, the system shouldn't wait for the second lock to be established, but remove the first lock, wait a (random) time, and start over locking again.

[4]You will need to depend on `liblockfile1 (>>1.01)` to use these functions.

12.7 News system configuration

All the configuration files related to the NNTP (news) servers and clients should be located under `/etc/news`.

There are some configuration issues that apply to a number of news clients and server packages on the machine. These are:

`/etc/news/organization` A string which should appear as the organization header for all messages posted by NNTP clients on the machine

`/etc/news/server` Contains the FQDN of the upstream NNTP server, or localhost if the local machine is an NNTP server.

Other global files may be added as required for cross-package news configuration.

12.8 Programs for the X Window System

12.8.1 Providing X support and package priorities

Programs that can be configured with support for the X Window System must be configured to do so and must declare any package dependencies necessary to satisfy their runtime requirements when using the X Window System. If such a package is of higher priority than the X packages on which it depends, it is required that either the X-specific components be split into a separate package, or that an alternative version of the package, which includes X support, be provided, or that the package's priority be lowered.

12.8.2 Packages providing an X server

Packages that provide an X server that, directly or indirectly, communicates with real input and display hardware should declare in their control data that they provide the virtual package `xserver`.[5]

12.8.3 Packages providing a terminal emulator

Packages that provide a terminal emulator for the X Window System which meet the criteria listed below should declare in their control data that they provide the virtual package `x-terminal-emulator`. They should also register themselves as an alternative for `/usr/bin/x-terminal-emulator`, with a priority of 20.

To be an `x-terminal-emulator`, a program must:

[5]This implements current practice, and provides an actual policy for usage of the `xserver` virtual package which appears in the virtual packages list. In a nutshell, X servers that interface directly with the display and input hardware or via another subsystem (e.g., GGI) should provide `xserver`. Things like `Xvfb`, `Xnest`, and `Xprt` should not.

- Be able to emulate a DEC VT100 terminal, or a compatible terminal.
- Support the command-line option `-e` *`command`*, which creates a new terminal window[6] and runs the specified *command*.
- Support the command-line option `-T` *`title`*, which creates a new terminal window with the window title *title*.

12.8.4 Packages providing a window manager

Packages that provide a window manager should declare in their control data that they provide the virtual package `x-window-manager`. They should also register themselves as an alternative for `/usr/bin/x-window-manager`, with a priority calculated as follows:

- Start with a priority of 20.
- If the window manager supports the Debian menu system, add 20 points if this support is available in the package's default configuration (i.e., no configuration files belonging to the system or user have to be edited to activate the feature); if configuration files must be modified, add only 10 points.
- If the window manager permits the X session to be restarted using a *different* window manager (without killing the X server) in its default configuration, add 10 points; otherwise add none.

12.8.5 Packages providing fonts

Packages that provide fonts for the X Window System[7] must do a number of things to ensure that they are both available without modification of the X or font server configuration, and that they do not corrupt files used by other font packages to register information about themselves.

1. Fonts of any type supported by the X Window System must be be in a separate binary package from any executables, libraries, or documentation (except that specific to the fonts shipped, such as their license information). If one or more of the fonts so packaged are necessary for proper operation of the package with which they are associated the font package may be Recommended; if the fonts merely provide an enhancement, a Suggests relationship may be used. Packages must not Depend on font packages.[8]

2. BDF fonts must be converted to PCF fonts with the `bdftopcf` utility (available in the `xutils` package, `gzip`ped, and placed in a directory that corresponds to their resolution:
   - 100 dpi fonts must be placed in `/usr/X11R6/lib/X11/fonts/100dpi/`.

[6] "New terminal window" does not necessarily mean a new top-level X window directly parented by the window manager; it could, if the terminal emulator application were so coded, be a new "view" in a multiple-document interface (MDI).

[7] For the purposes of Debian Policy, a "font for the X Window System" is one which is accessed via X protocol requests. Fonts for the Linux console, for PostScript renderers, or any other purpose, do not fit this definition. Any tool which makes such fonts available to the X Window System, however, must abide by this font policy.

[8] This is because the X server may retrieve fonts from the local filesystem or over the network from an X font server; the Debian package system is empowered to deal only with the local filesystem.

- 75 dpi fonts must be placed in `/usr/X11R6/lib/X11/fonts/75dpi/`.
- Character-cell fonts, cursor fonts, and other low-resolution fonts must be placed in `/usr/X11R6/lib/X11/fo`

3. Speedo fonts must be placed in `/usr/X11R6/lib/X11/fonts/Speedo/`.

4. Type 1 fonts must be placed in `/usr/X11R6/lib/X11/fonts/Type1/`. If font metric files are available, they must be placed here as well.

5. Subdirectories of `/usr/X11R6/lib/X11/fonts/` other than those listed above must be neither created nor used. (The `PEX`, `CID`, and `cyrillic` directories are excepted for historical reasons, but installation of files into these directories remains discouraged.)

6. Font packages may, instead of placing files directly in the X font directories listed above, provide symbolic links in the font directory which point to the files' actual location in the filesystem. Such a location must comply with the FHS.

7. Font packages should not contain both 75dpi and 100dpi versions of a font. If both are available, they should be provided in separate binary packages with `-75dpi` or `-100dpi` appended to the names of the packages containing the corresponding fonts.

8. Fonts destined for the `misc` subdirectory should not be included in the same package as 75dpi or 100dpi fonts; instead, they should be provided in a separate package with `-misc` appended to its name.

9. Font packages must not provide the files `fonts.dir`, `fonts.alias`, or `fonts.scale` in a font directory:

   - `fonts.dir` files must not be provided at all.
   - `fonts.alias` and `fonts.scale` files, if needed, should be provided in the directory `/etc/X11/fonts/f` where *fontdir* is the name of the subdirectory of `/usr/X11R6/lib/X11/fonts/` where the package's corresponding fonts are stored (e.g., `75dpi` or `misc`), *package* is the name of the package that provides these fonts, and *extension* is either `scale` or `alias`, whichever corresponds to the file contents.

10. Font packages must declare a dependency on `xutils (>> 4.0.3)` in their control data.

11. Font packages that provide one or more `fonts.scale` files as described above must invoke `update-fonts-scale` on each directory into which they installed fonts *before* invoking `update-fonts-dir` on that directory. This invocation must occur in both the `postinst` (for all arguments) and `postrm` (for all arguments except `upgrade`) scripts.

12. Font packages that provide one or more `fonts.alias` files as described above must invoke `update-fonts-alias` on each directory into which they installed fonts. This invocation must occur in both the `postinst` (for all arguments) and `postrm` (for all arguments except `upgrade`) scripts.

13. Font packages must invoke `update-fonts-dir` on each directory into which they installed fonts. This invocation must occur in both the `postinst` (for all arguments) and `postrm` (for all arguments except `upgrade`) scripts.

14. Font packages must not provide alias names for the fonts they include which collide with alias names already in use by fonts already packaged.

15. Font packages must not provide fonts with the same XLFD registry name as another font already packaged.

12.8.6 Application defaults files

Application defaults files must be installed in the directory `/etc/X11/app-defaults/` (use of a localized subdirectory of `/etc/X11/` as described in the *X Toolkit Intrinsics - C Language Interface* manual is also permitted). They must be registered as `conffiles` or handled as configuration files. Packages must not provide the directory `/usr/X11R6/lib/X11/app-defaults/`.

Customization of programs' X resources may also be supported with the provision of a file with the same name as that of the package placed in the `/etc/X11/Xresources/` directory, which must registered as a `conffile` or handled as a configuration file.[9] *Important:* packages that install files into the `/etc/X11/Xresources/` directory must conflict with `xbase (<< 3.3.2.3a-2)`; if this is not done it is possible for the installing package to destroy a previously-existing `/etc/X11/Xresources` file which had been customized by the system administrator.

12.8.7 Installation directory issues

Packages using the X Window System should not be configured to install files under the `/usr/X11R6/` directory unless they use `imake`. The `/usr/X11R6/` directory hierarchy should be regarded as deprecated for all packages except the X Window System itself, and those which use the `imake` program it provides, in which case the packages may transition out of the `/usr/X11R6/` directory at the maintainer's discretion.[10] Programs that use GNU `autoconf` and `automake` are usually easily configured at compile time to use `/usr/` instead of `/usr/X11R6/`, and this should be done whenever possible. Configuration files for window managers and display managers should be placed in a subdirectory of `/etc/X11/` corresponding to the package name due to these programs' tight integration with the mechanisms of the X Window System. Application-level programs should use the `/etc/` directory unless otherwise mandated by policy. The installation of files into subdirectories of `/usr/X11R6/include/X11/` and `/usr/X11R6/lib/X11/`

[9] Note that this mechanism is not the same as using app-defaults; app-defaults are tied to the client binary on the local filesystem, whereas X resources are stored in the X server and affect all connecting clients.

[10] `Imake`-using programs are exempt because, as long as they are written correctly, the pathnames they use to locate resources and install themselves are derived wholly from the X Window System configuration. Thus, in the event that the X Window System moves to `/usr/X11R7/`, `/usr/X12/`, or just plain `/usr/`, all that is required for these programs is a recompile against the corresponding X Window System library development packages.

is permitted but discouraged; package maintainers should determine if subdirectories of `/usr/lib/` and `/usr/share/` can be used instead. (The use of symbolic links from the `X11R6` directories to other FHS-compliant locations is encouraged if the program is not easily configured to look elsewhere for its files.) Packages must not provide or install files into the directories `/usr/bin/X11/`, `/usr/include/X11/` or `/usr/lib/X11/`. Files within a package should, however, make reference to these directories, rather than their `X11R6`-named counterparts `/usr/X11R6/bin/`, `/usr/X11R6/include/X11/` and `/usr/X11R6/lib/` if the resources being referred to have not been moved to other FHS-compliant locations.

12.8.8 The OSF/Motif and OpenMotif libraries

Programs that require the non-DFSG-compliant OSF/Motif or OpenMotif libraries[11] should be compiled against and tested with LessTif (a free re-implementation of Motif) instead. If the maintainer judges that the program or programs do not work sufficiently well with LessTif to be distributed and supported, but do so when compiled against Motif, then two versions of the package should be created; one linked statically against Motif and with `-smotif` appended to the package name, and one linked dynamically against Motif and with `-dmotif` appended to the package name. Both Motif-linked versions are dependent upon non-DFSG-compliant software and thus cannot be uploaded to the *main* distribution; if the software is itself DFSG-compliant it may be uploaded to the *contrib* distribution. While known existing versions of Motif permit unlimited redistribution of binaries linked against the library (whether statically or dynamically), it is the package maintainer's responsibility to determine whether this is permitted by the license of the copy of Motif in his or her possession.

12.9 Perl programs and modules

Perl programs and modules should follow the current Perl policy as defined in the file found on `ftp.debian.org` in `/debian/doc/package-developer/perl-policy.txt.gz` or your local mirror. In addition, it is included in the `debian-policy` package.

12.10 Emacs lisp programs

Please refer to the 'Debian Emacs Policy' (documented in `debian-emacs-policy.gz` of the `emacsen-common` package) for details of how to package emacs lisp programs.

12.11 Games

The permissions on `/var/games` are mode 755, owner `root` and group `root`.

[11] OSF/Motif and OpenMotif are collectively referred to as "Motif" in this policy document.

Each game decides on its own security policy.

Games which require protected, privileged access to high-score files, savegames, etc., may 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. (If an attacker can subvert any set-user-id game they can overwrite the executable of any other, causing other players of these games to run a Trojan horse program. 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.)

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 can only be accessed through set-id programs provided. You should 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.

As described in the FHS, binaries of games should be installed in the directory `/usr/games`. This also applies to games that use the X Window System. Manual pages for games (X and non-X games) should be installed in `/usr/share/man/man6`.

Chapter 13

Documentation

13.1 Manual pages

You should install manual pages in `nroff` source form, in appropriate places under `/usr/share/man`. You should only use sections 1 to 9 (see the FHS for more details). You must not install a preformatted 'cat page'.

Each program, utility, and function should have an associated manpage included in the same package. It is suggested that all configuration files also have a manual page included as well.

If no manual page is available for a particular program, utility, function or configuration file and this is reported as a bug to the Debian Bug Tracking System, a symbolic link from the requested manual page to the `undocumented(7)` manual page may be provided. This symbolic link can be created from `debian/rules` like this:

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

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.

Manual pages should be installed compressed using `gzip -9`.

If one manpage needs to be accessible 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. You should not create hard links in the manual page directories, nor 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/share/man`). If you do not create any links (whether symlinks, hard links, or `.so` directives) in the filesystem to the alternate names of the manpage, then you should not rely on `man` finding your manpage under those names based solely on the information in the manpage's header.[1]

13.2 Info documents

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

Your package should call `install-info` to update the Info `dir` file in its `postinst` script when called with a `configure` argument, for example:

```
install-info --quiet --section Development Development \
  /usr/share/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/share/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 should remove the entries in the `prerm` script when called with a `remove` argument:

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

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

13.3 Additional documentation

Any additional documentation that comes with the package may be installed at the discretion of the package maintainer. Text documentation should be installed in the directory `/usr/share/doc/`*`package`*, where *package* is the name of the package, 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.

[1] Supporting this in `man` often requires unreasonable processing time to find a manual page or to report that none exists, and moves knowledge into man's database that would be better left in the filesystem. This support is therefore deprecated and will cease to be present in the future.

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

Files in /usr/share/doc should not be referenced by any program, and the system administrator should be able to delete them without causing any programs to break. Any files that are referenced by programs but are also useful as standalone documentation should be installed under /usr/share/*package*/ with symbolic links from /usr/share/doc/*package*/.

13.4 Accessing the documentation

Former Debian releases placed all additional documentation in /usr/doc/*package*. To realize a smooth migration to /usr/share/doc/*package*, each package must maintain a symlink /usr/doc/*package* that points to the new location of its documentation in /usr/share/doc/*package*[2]. The symlink must be created when the package is installed; it cannot be contained in the package itself due to problems with dpkg. One reasonable way to accomplish this is to put the following in the package's postinst[3]:

```
if [ "$1" = "configure" ]; then
  if [ -d /usr/doc -a ! -e /usr/doc/package \
       -a -d /usr/share/doc/package ]; then
    ln -sf ../share/doc/package /usr/doc/package
  fi
fi
```

and the following in the package's prerm:

```
if [ \( "$1" = "upgrade" -o "$1" = "remove" \) \
     -a -L /usr/doc/package ]; then
  rm -f /usr/doc/package
fi
```

13.5 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 a binary package, in the directory /usr/share/doc/*approp package* or its subdirectories.[4]

[2]These symlinks will be removed in the future, but they have to be there for compatibility reasons until all packages have moved and the policy is changed accordingly.

[3]The debhelper script dh_installdocs does this automatically.

[4]The rationale: The important thing here is that HTML docs should be available in *some* package, not necessarily in the main binary package.

Other formats such as PostScript may be provided at the package maintainer's discretion.

13.6 Copyright information

Every package must be accompanied by a verbatim copy of its copyright and distribution license in the file `/usr/share/doc/`*package*`/copyright`. This file must neither be compressed nor be a symbolic link.

In addition, the copyright file must say where the upstream sources (if any) were obtained, and should explain briefly what modifications were made in the Debian version of the package compared to the upstream one. It should name the original authors of the package and the Debian maintainer(s) who were involved with its creation.

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

`/usr/share/doc/`*package* may be a symbolic link to another directory in `/usr/share/doc` only if the two packages both come from the same source and the first package Depends on the second. These rules are important because copyrights must be extractable by mechanical means.

Packages distributed under the UCB BSD license, the Artistic license, the GNU GPL, and the GNU LGPL should refer to the files `/usr/share/common-licenses/BSD`, `/usr/share/common-licenses/Artistic`, `/usr/share/common-licenses/GPL`, and `/usr/share/common-licenses/LGPL` respectively, rather than quoting them in the copyright file.

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

13.7 Examples

Any examples (configurations, source files, whatever), should be installed in a directory `/usr/share/doc/`*package*`/ex` These files should not be referenced by any program: they're there for the benefit of the system administrator and users as documentation only. Architecture-specific example files should be installed in a directory `/usr/lib/`*package*`/examples` with symbolic links to them from `/usr/share/doc/`*package*`/examples`, or the latter directory itself may be a symbolic link to the former.

13.8 Changelog files

Packages that are not Debian-native must contain a compressed copy of the `debian/changelog` file from the Debian source tree in `/usr/share/doc/`*package* with the name `changelog.Debian.gz`. If an upstream changelog is available, it should be accessible as `/usr/share/doc/`*`package`*`/changelog.gz` in plain text. If the upstream changelog is distributed in HTML, it should be made available in that form as `/usr/share/doc/`*`package`*`/changelog.html.gz` and a plain text `changelog.gz` should be generated from it using, for example, `lynx -dump -nolist`. If the upstream changelog files do not already conform to this naming convention, then this may be achieved either by renaming the files, or by adding a symbolic link, at the maintainer's discretion.[5]

All of these files 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/share/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`.

[5]Rationale: People should not have to look in places for upstream changelogs merely because they are given different names or are distributed in HTML format.

Appendix A

Introduction and scope of these appendices

These appendices are taken essentially verbatim from the now-deprecated Packaging Manual, version 3.2.1.0. They are the chapters which are likely to be of use to package maintainers and which have not already been included in the policy document itself. Most of these sections are very likely not relevant to policy; they should be treated as documentation for the packaging system. Please note that these appendices are included for convenience, and for historical reasons: they used to be part of policy package, and they have not yet been incorporated into dpkg documentation. However, they still have value, and hence they are presented here.

They have not yet been checked to ensure that they are compatible with the contents of policy, and if there are any contradictions, the version in the main policy document takes precedence. The remaining chapters of the old Packaging Manual have also not been read in detail to ensure that there are not parts which have been left out. Both of these will be done in due course.

`dpkg` is a suite of programs for creating binary package files and installing and removing them on Unix systems.[1]

The binary packages are designed for the management of installed executable programs (usually compiled binaries) and their associated data, though source code examples and documentation are provided as part of some packages.

This manual describes the technical aspects of creating Debian binary packages (`.deb` files). It documents the behaviour of the package management programs `dpkg`, `dselect` et al. and the way they interact with packages.

It also documents the interaction between `dselect`'s core and the access method scripts it uses to actually install the selected packages, and describes how to create a new access method.

This manual does not go into detail about the options and usage of the package building and installation tools. It should therefore be read in conjuction with those programs' manpages.

[1] `dpkg` is targetted primarily at Debian GNU/Linux, but may work on or be ported to other systems.

The utility programs which are provided with `dpkg` for managing various system configuration and similar issues, such as `update-rc.d` and `install-info`, are not described in detail here - please see their manpages.

It does *not* describe the policy requirements imposed on Debian packages, such as the permissions on files and directories, documentation requirements, upload procedure, and so on. You should see the Debian packaging policy manual for these details. (Many of them will probably turn out to be helpful even if you don't plan to upload your package and make it available as part of the distribution.)

It is assumed that the reader is reasonably familiar with the `dpkg` System Administrators' manual. Unfortunately this manual does not yet exist.

The Debian version of the FSF's GNU hello program is provided as an example for people wishing to create Debian packages. The Debian `debmake` package is recommended as a very helpful tool in creating and maintaining Debian packages. However, while the tools and examples are helpful, they do not replace the need to read and follow the Policy and Programmer's Manual.

Appendix B

Binary packages (from old Packaging Manual)

The binary package has two main sections. The first part consists of various control information files and scripts used by `dpkg` when installing and removing. See 'Package control information files' on the next page.

The second part is an archive containing the files and directories to be installed.

In the future binary packages may also contain other components, such as checksums and digital signatures. The format for the archive is described in full in the `deb(5)` manpage.

B.1 Creating package files - `dpkg-deb`

All manipulation of binary package files is done by `dpkg-deb`; it's the only program that has knowledge of the format. (`dpkg-deb` may be invoked by calling `dpkg`, as `dpkg` will spot that the options requested are appropriate to `dpkg-deb` and invoke that instead with the same arguments.)

In order to create a binary package you must make a directory tree which contains all the files and directories you want to have in the filesystem data part of the package. In Debian-format source packages this directory is usually `debian/tmp`, relative to the top of the package's source tree.

They should have the locations (relative to the root of the directory tree you're constructing) ownerships and permissions which you want them to have on the system when they are installed.

With current versions of `dpkg` the uid/username and gid/groupname mappings for the users and groups being used should be the same on the system where the package is built and the one where it is installed.

You need to add one special directory to the root of the miniature filesystem tree you're creating: `DEBIAN`. It should contain the control information files, notably the binary package control file (see 'The main control information file: `control`' on page 99).

The `DEBIAN` directory will not appear in the filesystem archive of the package, and so won't be installed by `dpkg` when the package is installed.

When you've prepared the package, you should invoke:

```
dpkg --build directory
```

This will build the package in *`directory`*`.deb`. (`dpkg` knows that `--build` is a `dpkg-deb` option, so it invokes `dpkg-deb` with the same arguments to build the package.)

See the manpage `dpkg-deb(8)` for details of how to examine the contents of this newly-created file. You may find the output of following commands enlightening:

```
dpkg-deb --info filename.deb
dpkg-deb --contents filename.deb
dpkg --contents filename.deb
```

To view the copyright file for a package you could use this command:

```
dpkg --fsys-tarfile filename.deb | tar xof usr/share/doc/\*copyright | less
```

B.2 Package control information files

The control information portion of a binary package is a collection of files with names known to `dpkg`. It will treat the contents of these files specially - some of them contain information used by `dpkg` when installing or removing the package; others are scripts which the package maintainer wants `dpkg` to run.

It is possible to put other files in the package control area, but this is not generally a good idea (though they will largely be ignored).

Here is a brief list of the control info files supported by `dpkg` and a summary of what they're used for.

`control` This is the key description file used by `dpkg`. It specifies the package's name and version, gives its description for the user, states its relationships with other packages, and so forth. See 'The main control information file: `control`' on the next page.

It is usually generated automatically from information in the source package by the `dpkg-gencontrol` program, and with assistance from `dpkg-shlibdeps`. See 'Tools for processing source packages' on page 101.

postinst, preinst, postrm, prerm These are exectuable files (usually scripts) which dpkg runs during installation, upgrade and removal of packages. They allow the package to deal with matters which are particular to that package or require more complicated processing than that provided by dpkg. Details of when and how they are called are in 'Package maintainer scripts and installation procedure' on page 31.

It is very important to make these scripts idempotent. [1] This is so that if an error occurs, the user interrupts dpkg or some other unforeseen circumstance happens you don't leave the user with a badly-broken package.

The maintainer scripts are guaranteed to run with a controlling terminal and can interact with the user. If they need to prompt for passwords, do full-screen interaction or something similar you should do these things to and from /dev/tty, since dpkg will at some point redirect scripts' standard input and output so that it can log the installation process. Likewise, because these scripts may be executed with standard output redirected into a pipe for logging purposes, Perl scripts should set unbuffered output by setting $|=1 so that the output is printed immediately rather than being buffered.

Each script should return a zero exit status for success, or a nonzero one for failure.

conffiles This file contains a list of configuration files which are to be handled automatically by dpkg (see 'Configuration file handling (from old Packaging Manual)' on page 123). Note that not necessarily every configuration file should be listed here.

shlibs This file contains a list of the shared libraries supplied by the package, with dependency details for each. This is used by dpkg-shlibdeps when it determines what dependencies are required in a package control file. The shlibs file format is described on 'The shlibs File Format' on page 50.

B.3 The main control information file: control

The most important control information file used by dpkg when it installs a package is control. It contains all the package's 'vital statistics'.

The binary package control files of packages built from Debian sources are made by a special tool, dpkg-gencontrol, which reads debian/control and debian/changelog to find the information it needs. See 'Source packages (from old Packaging Manual)' on page 101 for more details.

The fields in binary package control files are:

- Package (mandatory)
- Version (mandatory)
- Architecture (mandatory) [2]
- Depends, Provides et al.

[1] That means that if it runs successfully or fails and then you call it again it doesn't bomb out, but just ensures that everything is the way it ought to be.

[2] This field should appear in all packages, though dpkg doesn't require it yet so that old packages can still be installed.

- `Essential`
- `Maintainer`
- `Section`, `Priority`
- `Source`
- `Description`
- `Installed-Size`

A description of the syntax of control files and the purpose of these fields is available in 'Control files and their fields (from old Packaging Manual)' on page 115.

B.4 Time Stamps

Maintainers are encouraged to preserve the modification times of the upstream source files in a package, as far as is reasonably possible. [3]

[3] The rationale is that there is some information conveyed by knowing the age of the file, for example, you could recognize that some documentation is very old by looking at the modification time, so it would be nice if the modification time of the upstream source would be preserved.

Appendix C

Source packages (from old Packaging Manual)

The Debian binary packages in the distribution are generated from Debian sources, which are in a special format to assist the easy and automatic building of binaries.

There was a previous version of the Debian source format, which is now being phased out. Instructions for converting an old-style package are given in the Debian policy manual.

C.1 Tools for processing source packages

Various tools are provided for manipulating source packages; they pack and unpack sources and help build of binary packages and help manage the distribution of new versions.

They are introduced and typical uses described here; see `dpkg-source(1)` for full documentation about their arguments and operation.

For examples of how to construct a Debian source package, and how to use those utilities that are used by Debian source packages, please see the `hello` example package.

C.1.1 `dpkg-source` - packs and unpacks Debian source packages

This program is frequently used by hand, and is also called from package-independent automated building scripts such as `dpkg-buildpackage`.

To unpack a package it is typically invoked with

```
dpkg-source -x .../path/to/filename.dsc
```

with the *filename*.tar.gz and *filename*.diff.gz (if applicable) in the same directory. It unpacks into *package-version*, and if applicable *package-version*.orig, in the current directory.

To create a packed source archive it is typically invoked:

```
dpkg-source -b package-version
```

This will create the .dsc, .tar.gz and .diff.gz (if appropriate) in the current directory. dpkg-source does not clean the source tree first - this must be done separately if it is required.

See also 'Source packages as archives' on page 112.

C.1.2 dpkg-buildpackage - overall package-building control script

dpkg-buildpackage is a script which invokes dpkg-source, the debian/rules targets clean, build and binary, dpkg-genchanges and pgp to build a signed source and binary package upload.

It is usually invoked by hand from the top level of the built or unbuilt source directory. It may be invoked with no arguments; useful arguments include:

-uc, -us Do not PGP-sign the .changes file or the source package .dsc file, respectively.

-p*pgp-command* Invoke *pgp-command* instead of finding pgp on the PATH. *pgp-command* must behave just like pgp.

-r*root-command* When root privilege is required, invoke the command *root-command*. *root-command* should invoke its first argument as a command, from the PATH if necessary, and pass its second and subsequent arguments to the command it calls. If no *root-command* is supplied then *dpkg-buildpackage* will take no special action to gain root privilege, so that for most packages it will have to be invoked as root to start with.

-b, -B Two types of binary-only build and upload - see dpkg-source(1).

C.1.3 dpkg-gencontrol - generates binary package control files

This program is usually called from debian/rules (see 'The Debianised source tree' on page 105) in the top level of the source tree.

This is usually done just before the files and directories in the temporary directory tree where the package is being built have their permissions and ownerships set and the package is constructed using dpkg-deb/ [1].

dpkg-gencontrol must be called after all the files which are to go into the package have been placed in the temporary build directory, so that its calculation of the installed size of a package is correct.

It is also necessary for dpkg-gencontrol to be run after dpkg-shlibdeps so that the variable substitutions created by dpkg-shlibdeps in debian/substvars are available.

[1] This is so that the control file which is produced has the right permissions

For a package which generates only one binary package, and which builds it in `debian/tmp` relative to the top of the source package, it is usually sufficient to call `dpkg-gencontrol`.

Sources which build several binaries will typically need something like:

```
dpkg-gencontrol -Pdebian/tmp-pkg -ppackage
```

The `-P` tells `dpkg-gencontrol` that the package is being built in a non-default directory, and the `-p` tells it which package's control file should be generated.

`dpkg-gencontrol` also adds information to the list of files in `debian/files`, for the benefit of (for example) a future invocation of `dpkg-genchanges`.

C.1.4 `dpkg-shlibdeps` - calculates shared library dependencies

This program is usually called from `debian/rules` just before `dpkg-gencontrol` (see 'The Debianised source tree' on page 105), in the top level of the source tree.

Its arguments are executables. [2] for which shared library dependencies should be included in the binary package's control file.

If some of the found shared libraries should only warrant a `Recommends` or `Suggests`, or if some warrant a `Pre-Depends`, this can be achieved by using the `-ddependency-field` option before those executable(s). (Each `-d` option takes effect until the next `-d`.)

`dpkg-shlibdeps` does not directly cause the output control file to be modified. Instead by default it adds to the `debian/substvars` file variable settings like `shlibs:Depends`. These variable settings must be referenced in dependency fields in the appropriate per-binary-package sections of the source control file.

For example, the `procps` package generates two kinds of binaries, simple C binaries like `ps` which require a predependency and full-screen ncurses binaries like `top` which require only a recommendation. It can say in its `debian/rules`:

```
dpkg-shlibdeps -dPre-Depends ps -dRecommends top
```

and then in its main control file `debian/control`:

```
...
Package: procps
Pre-Depends: ${shlibs:Pre-Depends}
Recommends: ${shlibs:Recommends}
...
```

[2] In a forthcoming dpkg version, `dpkg-shlibdeps` would be required to be called on shared libraries as well. They may be specified either in the locations in the source tree where they are created or in the locations in the temporary build tree where they are installed prior to binary package creation.

Sources which produce several binary packages with different shared library dependency requirements can use the `-p`*`varnameprefix`* option to override the default `shlib:` prefix (one invocation of `dpkg-shlibdeps` per setting of this option). They can thus produce several sets of dependency variables, each of the form *`varnameprefix`*`:`*`dependencyfield`*, which can be referred to in the appropriate parts of the binary package control files.

C.1.5 `dpkg-distaddfile` - adds a file to `debian/files`

Some packages' uploads need to include files other than the source and binary package files.

`dpkg-distaddfile` adds a file to the `debian/files` file so that it will be included in the `.changes` file when `dpkg-genchanges` is run.

It is usually invoked from the `binary` target of `debian/rules`:

```
dpkg-distaddfile filename section priority
```

The *filename* is relative to the directory where `dpkg-genchanges` will expect to find it - this is usually the directory above the top level of the source tree. The `debian/rules` target should put the file there just before or just after calling `dpkg-distaddfile`.

The *section* and *priority* are passed unchanged into the resulting `.changes` file. See '`Section` and `Priority`' on page 118.

C.1.6 `dpkg-genchanges` - generates a `.changes` upload control file

This program is usually called by package-independent automatic building scripts such as `dpkg-buildpackage`, but it may also be called by hand.

It is usually called in the top level of a built source tree, and when invoked with no arguments will print out a straightforward `.changes` file based on the information in the source package's changelog and control file and the binary and source packages which should have been built.

C.1.7 `dpkg-parsechangelog` - produces parsed representation of a changelog

This program is used internally by `dpkg-source` et al. It may also occasionally be useful in `debian/rules` and elsewhere. It parses a changelog, `debian/changelog` by default, and prints a control-file format representation of the information in it to standard output.

C.1.8 `dpkg-architecture` - information about the build and host system

This program can be used manually, but is also invoked by `dpkg-buildpackage` or `debian/rules` to set to set environment or make variables which specify the build and host architecture for the package building process.

C.2 The Debianised source tree

The source archive scheme described later is intended to allow a Debianised source tree with some associated control information to be reproduced and transported easily. The Debianised source tree is a version of the original program with certain files added for the benefit of the Debianisation process, and with any other changes required made to the rest of the source code and installation scripts.

The extra files created for Debian are in the subdirectory `debian` of the top level of the Debianised source tree. They are described below.

C.2.1 `debian/rules` - the main building script

This file is an executable makefile, and contains the package-specific recipies for compiling the package and building binary package(s) out of the source.

It must start with the line `#!/usr/bin/make -f`, so that it can be invoked by saying its name rather than invoking `make` explicitly.

Since an interactive `debian/rules` script makes it impossible to autocompile that package and also makes it hard for other people to reproduce the same binary package, all **required targets** have to be non-interactive. At a minimul, required targets are the ones called by `dpkg-buildpackage`, namely, *clean*, *binary*, *binary-arch*, and *build*. It also follows that any target that these targets depend on must also be non-interactive.

The targets which are required to be present are:

`build` This should perform all non-interactive configuration and compilation of the package. If a package has an interactive pre-build configuration routine, the Debianised source package should be built after this has taken place, so that it can be built without rerunning the configuration.

A package may also provide both of the targets `build-arch` and `build-indep`. The `build-arch` target, if provided, should perform all non-interactive configuration and compilation required for producing all architecture-dependant binary packages (those packages for which the body of the `Architecture` field in `debian/control` is not `all`). Similarly, the `build-indep` target, if provided, should perform all non-interactive configuration and compilation required for producing all architecture-independent binary packages (those packages for which the body of the `Architecture` field in `debian/control` is `all`). The `build` target should depend on those of the targets `build-arch` and `build-indep` that are provided in the rules file.

If one or both of the targets `build-arch` and `build-indep` are not provided, then invoking `debian/rules` with one of the not-provided targets as arguments should produce a exit status code of 2. Usually this is provided automatically by make if the target is missing.

For some packages, notably ones where the same source tree is compiled in different ways to produce two binary packages, the `build` target does not make much sense. For these packages it is good enough to provide two (or more) targets (`build-a` and `build-b` or whatever) for each of the ways of building the package, and a `build` target that does nothing. The `binary` target will have to build the package in each of the possible ways and make the binary package out of each.

The targets `build`, `build-arch` and `build-indep` target must not do anything that might require root privilege.

The `build` target may need to run `clean` first - see below.

When a package has a configuration routine that takes a long time, or when the makefiles are poorly designed, or when `build` needs to run `clean` first, it is a good idea to `touch build` when the build process is complete. This will ensure that if `debian/rules build` is run again it will not rebuild the whole program.

`binary`, `binary-arch`, `binary-indep` The `binary` target should be all that is necessary for the user to build the binary package. All these targets are required to be non-interactive. It is split into two parts: `binary-arch` builds the packages' output files which are specific to a particular architecture, and `binary-indep` builds those which are not.

`binary` should usually be a target with no commands which simply depends on `binary-arch` and `binary-indep`.

Both `binary-*` targets should depend on the `build` target, above, so that the package is built if it has not been already. It should then create the relevant binary package(s), using `dpkg-gencontrol` to make their control files and `dpkg-deb` to build them and place them in the parent of the top level directory.

If one of the `binary-*` targets has nothing to do (this will be always be the case if the source generates only a single binary package, whether architecture-dependent or not) it *must* still exist, but should always succeed.

'Binary packages (from old Packaging Manual)' on page 97 describes how to construct binary packages.

The `binary` targets must be invoked as root.

`clean` This should undo any effects that the `build` and `binary` targets may have had, except that it should leave alone any output files created in the parent directory by a run of `binary`. This target is required to be non-interactive.

If a `build` file is touched at the end of the `build` target, as suggested above, it must be removed as the first thing that `clean` does, so that running `build` again after an interrupted `clean` doesn't think that everything is already done.

The `clean` target must be invoked as root if `binary` has been invoked since the last `clean`, or if `build` has been invoked as root (since `build` may create directories, for example).

`get-orig-source` (optional) This target fetches the most recent version of the original source package from a canonical archive site (via FTP or WWW, for example), does any necessary rearrangement to turn it into the original source tarfile format described below, and leaves it in the current directory.

This target may be invoked in any directory, and should take care to clean up any temporary files it may have left.

This target is optional, but providing it if possible is a good idea.

The `build`, `binary` and `clean` targets must be invoked with a current directory of the package's top-level directory.

Additional targets may exist in `debian/rules`, either as published or undocumented interfaces or for the package's internal use.

The architecture we build on and build for is determined by make variables via dpkg-architecture (see '`dpkg-architecture` - information about the build and host system' on page 105). You can get the Debian architecture and the GNU style architecture specification string for the build machine as well as the host machine. Here is a list of supported make variables:

- `DEB_*_ARCH` (the Debian architecture)
- `DEB_*_GNU_TYPE` (the GNU style architecture specification string)
- `DEB_*_GNU_CPU` (the CPU part of DEB_*_GNU_TYPE)
- `DEB_*_GNU_SYSTEM` (the System part of DEB_*_GNU_TYPE)

where `*` is either `BUILD` for specification of the build machine or `HOST` for specification of the machine we build for.

Backward compatibility can be provided in the rules file by setting the needed variables to suitable default values, please refer to the documentation of dpkg-architecture for details.

It is important to understand that the `DEB_*_ARCH` string does only determine which Debian architecture we build on resp. for. It should not be used to get the CPU or System information, the GNU style variables should be used for that.

C.2.2 `debian/control`

This file contains version-independent details about the source package and about the binary packages it creates.

It is a series of sets of control fields, each syntactically similar to a binary package control file. The sets are separated by one or more blank lines. The first set is information about the source package in general; each subsequent set describes one binary package that the source tree builds.

The syntax and semantics of the fields are described below in 'Control files and their fields (from old Packaging Manual)' on page 115.

The general (binary-package-independent) fields are:

- `Source` (mandatory)
- `Maintainer`
- `Section` and `Priority` (classification, mandatory)
- `Build-Depends` et al. (source package interrelationships)
- `Standards-Version`

The per-binary-package fields are:

- `Package` (mandatory)
- `Architecture` (mandatory)
- `Description`
- `Section` and `Priority` (classification)
- `Essential`
- `Depends` et al. (binary package interrelationships)

These fields are used by `dpkg-gencontrol` to generate control files for binary packages (see below), by `dpkg-genchanges` to generate the `.changes` file to accompany the upload, and by `dpkg-source` when it creates the `.dsc` source control file as part of a source archive.

The fields here may contain variable references - their values will be substituted by `dpkg-gencontrol`, `dpkg-genchanges` or `dpkg-source` when they generate output control files. See '`debian/substvars` and variable substitutions' on page 111 for details.

User-defined fields

Additional user-defined fields may be added to the source package control file. Such fields will be ignored, and not copied to (for example) binary or source package control files or upload control files.

If you wish to add additional unsupported fields to these output files you should use the mechanism described here.

Fields in the main source control information file with names starting `X`, followed by one or more of the letters `BCS` and a hyphen `-`, will be copied to the output files. Only the part of the field name after the hyphen will be used in the output file. Where the letter `B` is used the field will appear in binary package control files, where the letter `S` is used in source package control files and where `C` is used in upload control (`.changes`) files.

For example, if the main source information control file contains the field

```
XBS-Comment: I stand between the candle and the star.
```

then the binary and source package control files will contain the field

```
Comment: I stand between the candle and the star.
```

C.2.3 debian/changelog

This file records the changes to the Debian-specific parts of the package [3].

It has a special format which allows the package building tools to discover which version of the package is being built and find out other release-specific information.

That format is a series of entries like this:

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

 * change details
 more change details
 * even more change details

-- maintainer name and email address  date
```

package and *version* are the source package name and version number.

distribution(s) lists the distributions where this version should be installed when it is uploaded - it is copied to the `Distribution` field in the `.changes` file. See '`Distribution`' on page 120.

urgency is the value for the `Urgency` field in the `.changes` file for the upload. See '`Urgency`' on page 121. It is not possible to specify an urgency containing commas; commas are used to separate *`keyword`*`=`*`value`* settings in the `dpkg` changelog format (though there is currently only one useful *keyword*, `urgency`).

The change details may in fact be any series of lines starting with at least two spaces, but conventionally each change starts with an asterisk and a separating space and continuation lines are indented so as to bring them in line with the start of the text above. Blank lines may be used here to separate groups of changes, if desired.

The maintainer name and email address should *not* necessarily be those of the usual package maintainer. They should be the details of the person doing *this* version. The information here will be copied to the `.changes` file, and then later used to send an acknowledgement when the upload has been installed.

[3] Though there is nothing stopping an author who is also the Debian maintainer from using it for all their changes, it will have to be renamed if the Debian and upstream maintainers become different people.

The *date* should be in RFC822 format [4]; it should include the timezone specified numerically, with the timezone name or abbreviation optionally present as a comment.

The first 'title' line with the package name should start at the left hand margin; the 'trailer' line with the maintainer and date details should be preceded by exactly one space. The maintainer details and the date must be separated by exactly two spaces.

An Emacs mode for editing this format is available: it is called `debian-changelog-mode`. You can have this mode selected automatically when you edit a Debian changelog by adding a local variables clause to the end of the changelog.

Defining alternative changelog formats

It is possible to use a different format to the standard one, by providing a parser for the format you wish to use.

In order to have `dpkg-parsechangelog` run your parser, you must include a line within the last 40 lines of your file matching the Perl regular expression: `\schangelog-format:\s+([0-9a-z]+)\W` The part in parentheses should be the name of the format. For example, you might say:

```
@@@ changelog-format: joebloggs @@@
```

Changelog format names are non-empty strings of alphanumerics.

If such a line exists then `dpkg-parsechangelog` will look for the parser as `/usr/lib/dpkg/parsechangelog/f`
`name` or `/usr/local/lib/dpkg/parsechangelog/`*`format-name`*; it is an error for it not to find it, or for it not to be an executable program. The default changelog format is `dpkg`, and a parser for it is provided with the `dpkg` package.

The parser will be invoked with the changelog open on standard input at the start of the file. It should read the file (it may seek if it wishes) to determine the information required and return the parsed information to standard output in the form of a series of control fields in the standard format. By default it should return information about only the most recent version in the changelog; it should accept a `-v`*`version`* option to return changes information from all versions present *strictly after version*, and it should then be an error for *version* not to be present in the changelog.

The fields are:

- `Source`
- `Version` (mandatory)
- `Distribution` (mandatory)
- `Urgency` (mandatory)
- `Maintainer` (mandatory)

[4]This is generated by the `822-date` program.

- `Date`
- `Changes` (mandatory)

If several versions are being returned (due to the use of `-v`), the urgency value should be of the highest urgency code listed at the start of any of the versions requested followed by the concatenated (space-separated) comments from all the versions requested; the maintainer, version, distribution and date should always be from the most recent version.

For the format of the `Changes` field see ‘`Changes`’ on page 121.

If the changelog format which is being parsed always or almost always leaves a blank line between individual change notes these blank lines should be stripped out, so as to make the resulting output compact.

If the changelog format does not contain date or package name information this information should be omitted from the output. The parser should not attempt to synthesise it or find it from other sources.

If the changelog does not have the expected format the parser should exit with a nonzero exit status, rather than trying to muddle through and possibly generating incorrect output.

A changelog parser may not interact with the user at all.

C.2.4 `debian/substvars` and variable substitutions

When `dpkg-gencontrol`, `dpkg-genchanges` and `dpkg-source` generate control files they do variable substitutions on their output just before writing it. Variable substitutions have the form `${`*`variable-name`*`}`. The optional file `debian/substvars` contains variable substitutions to be used; variables can also be set directly from `debian/rules` using the `-V` option to the source packaging commands, and certain predefined variables are available.

The is usually generated and modified dynamically by `debian/rules` targets; in this case it must be removed by the `clean` target.

See `dpkg-source(1)` for full details about source variable substitutions, including the format of `debian/substvars`.

C.2.5 `debian/files`

This file is not a permanent part of the source tree; it is used while building packages to record which files are being generated. `dpkg-genchanges` uses it when it generates a `.changes` file.

It should not exist in a shipped source package, and so it (and any backup files or temporary files such as `files.new` [5]) should be removed by the `clean` target. It may also be wise to ensure a fresh start by emptying or removing it at the start of the `binary` target.

[5] `files.new` is used as a temporary file by `dpkg-gencontrol` and `dpkg-distaddfile` - they write a new version of `files` here before renaming it, to avoid leaving a corrupted copy if an error occurs

`dpkg-gencontrol` adds an entry to this file for the `.deb` file that will be created by `dpkg-deb` from the control file that it generates, so for most packages all that needs to be done with this file is to delete it in `clean`.

If a package upload includes files besides the source package and any binary packages whose control files were made with `dpkg-gencontrol` then they should be placed in the parent of the package's top-level directory and `dpkg-distaddfile` should be called to add the file to the list in `debian/files`.

C.2.6 `debian/tmp`

This is the canonical temporary location for the construction of binary packages by the `binary` target. The directory `tmp` serves as the root of the filesystem tree as it is being constructed (for example, by using the package's upstream makefiles install targets and redirecting the output there), and it also contains the `DEBIAN` subdirectory. See 'Creating package files - `dpkg-deb`' on page 97.

If several binary packages are generated from the same source tree it is usual to use several `debian/tmp`*`something`* directories, for example `tmp-a` or `tmp-doc`.

Whatever `tmp` directories are created and used by `binary` must of course be removed by the `clean` target.

C.3 Source packages as archives

As it exists on the FTP site, a Debian source package consists of three related files. You must have the right versions of all three to be able to use them.

Debian source control file - `.dsc` This file contains a series of fields, identified and separated just like the fields in the control file of a binary package. The fields are listed below; their syntax is described above, in 'Control files and their fields (from old Packaging Manual)' on page 115.

- `Source`
- `Version`
- `Maintainer`
- `Binary`
- `Architecture`
- `Build-Depends` et al. (source package interrelationships)
- `Standards-Version`
- `Files`

The source package control file is generated by `dpkg-source` when it builds the source archive, from other files in the source package, described above. When unpacking it is checked against the files and directories in the other parts of the source package, as described below.

Original source archive - *package_upstream-version*.orig.tar.gz This is a compressed (with `gzip -9`) `tar` file containing the source code from the upstream authors of the program. The tarfile unpacks into a directory *package-upstream-version*.orig, and does not contain files anywhere other than in there or in its subdirectories.

Debianisation diff - *package_upstream_version-revision*.diff.gz This is a unified context diff (`diff -u`) giving the changes which are required to turn the original source into the Debian source. These changes may only include editing and creating plain files. The permissions of files, the targets of symbolic links and the characteristics of special files or pipes may not be changed and no files may be removed or renamed.

All the directories in the diff must exist, except the `debian` subdirectory of the top of the source tree, which will be created by `dpkg-source` if necessary when unpacking.

The `dpkg-source` program will automatically make the `debian/rules` file executable (see below).

If there is no original source code - for example, if the package is specially prepared for Debian or the Debian maintainer is the same as the upstream maintainer - the format is slightly different: then there is no diff, and the tarfile is named *package_version*.tar.gz and contains a directory *package-version*.

C.4 Unpacking a Debian source package without dpkg-source

`dpkg-source -x` is the recommended way to unpack a Debian source package. However, if it is not available it is possible to unpack a Debian source archive as follows:

1. Untar the tarfile, which will create a `.orig` directory.
2. Rename the `.orig` directory to *package-version*.
3. Create the subdirectory `debian` at the top of the source tree.
4. Apply the diff using `patch -p0`.
5. Untar the tarfile again if you want a copy of the original source code alongside the Debianised version.

It is not possible to generate a valid Debian source archive without using `dpkg-source`. In particular, attempting to use `diff` directly to generate the `.diff.gz` file will not work.

C.4.1 Restrictions on objects in source packages

The source package may not contain any hard links [6] [7], device special files, sockets or setuid or setgid files. [8]

[6] This is not currently detected when building source packages, but only when extracting them.

[7] Hard links may be permitted at some point in the future, but would require a fair amount of work.

[8] Setgid directories are allowed.

The source packaging tools manage the changes between the original and Debianised source using `diff` and `patch`. Turning the original source tree as included in the `.orig.tar.gz` into the debianised source must not involve any changes which cannot be handled by these tools. Problematic changes which cause `dpkg-source` to halt with an error when building the source package are:

- Adding or removing symbolic links, sockets or pipes.
- Changing the targets of symbolic links.
- Creating directories, other than `debian`.
- Changes to the contents of binary files.

Changes which cause `dpkg-source` to print a warning but continue anyway are:

- Removing files, directories or symlinks. [9]
- Changed text files which are missing the usual final newline (either in the original or the modified source tree).

Changes which are not represented, but which are not detected by `dpkg-source`, are:

- Changing the permissions of files (other than `debian/rules`) and directories.

The `debian` directory and `debian/rules` are handled specially by `dpkg-source` - before applying the changes it will create the `debian` directory, and afterwards it will make `debian/rules` world-exectuable.

[9] Renaming a file is not treated specially - it is seen as the removal of the old file (which generates a warning, but is otherwise ignored), and the creation of the new one.

Appendix D

Control files and their fields (from old Packaging Manual)

Many of the tools in the dpkg suite manipulate data in a common format, known as control files. Binary and source packages have control data as do the .changes files which control the installation of uploaded files, and dpkg's internal databases are in a similar format.

D.1 Syntax of control files

A file consists of one or more paragraphs of fields. The paragraphs are separated by blank lines. Some control files only allow one paragraph; others allow several, in which case each paragraph often refers to a different package.

Each paragraph is a series of fields and values; each field consists of a name, followed by a colon and the value. It ends at the end of the line. Horizontal whitespace (spaces and tabs) may occur before or after the value and is ignored there; it is conventional to put a single space after the colon.

Some fields' values may span several lines; in this case each continuation line *must* start with a space or tab. Any trailing spaces or tabs at the end of individual lines of a field value are ignored.

Except where otherwise stated only a single line of data is allowed and whitespace is not significant in a field body. Whitespace may never appear inside names (of packages, architectures, files or anything else), version numbers or in between the characters of multi-character version relationships.

Field names are not case-sensitive, but it is usual to capitalise the field names using mixed case as shown below.

Blank lines, or lines consisting only of spaces and tabs, are not allowed within field values or between fields - that would mean a new paragraph.

It is important to note that there are several fields which are optional as far as `dpkg` and the related tools are concerned, but which must appear in every Debian package, or whose omission may cause problems. When writing the control files for Debian packages you *must* read the Debian policy manual in conjuction with the details below and the list of fields for the particular file.

D.2 List of fields

D.2.1 `Package`

The name of the binary package. Package names consist of the alphanumerics and `+ - .` (plus, minus and full stop). [1]

They must be at least two characters and must start with an alphanumeric. In current versions of dpkg they are sort of case-sensitive[2]; use lowercase package names unless the package you're building (or referring to, in other fields) is already using uppercase.

D.2.2 `Version`

This lists the source or binary package's version number - see 'Version numbering' on page 19.

D.2.3 `Architecture`

This is the architecture string; it is a single word for the Debian architecture.

`dpkg` will check the declared architecture of a binary package against its own compiled-in value before it installs it.

The special value `all` indicates that the package is architecture-independent.

In the main `debian/control` file in the source package, or in the source package control file `.dsc`, a list of architectures (separated by spaces) is also allowed, as is the special value `any`. A list indicates that the source will build an architecture-dependent package, and will only work correctly on the listed architectures. `any` indicates that though the source package isn't dependent on any particular architecture and should compile fine on any one, the binary package(s) produced are not architecture-independent but will instead be specific to whatever the current build architecture is.

In a `.changes` file the `Architecture` field lists the architecture(s) of the package(s) currently being uploaded. This will be a list; if the source for the package is being uploaded too the special entry `source` is also present.

[1] The characters `@ : = %` `_` (at, colon, equals, percent and underscore) used to be legal and are still accepted when found in a package file, but may not be used in new packages

[2] This is a bug.

See '`debian/rules` - the main building script' on page 105 for information how to get the architecture for the build process.

D.2.4 `Maintainer`

The package maintainer's name and email address. The name should come first, then the email address inside angle brackets <> (in RFC822 format).

If the maintainer's name contains a full stop then the whole field will not work directly as an email address due to a misfeature in the syntax specified in RFC822; a program using this field as an address must check for this and correct the problem if necessary (for example by putting the name in round brackets and moving it to the end, and bringing the email address forward).

In a `.changes` file or parsed changelog data this contains the name and email address of the person responsible for the particular version in question - this may not be the package's usual maintainer.

This field is usually optional in as far as the `dpkg` are concerned, but its absence when building packages usually generates a warning.

D.2.5 `Source`

This field identifies the source package name.

In a main source control information or a `.changes` or `.dsc` file or parsed changelog data this may contain only the name of the source package.

In the control file of a binary package (or in a `Packages` file) it may be followed by a version number in parentheses. [3] This version number may be omitted (and is, by `dpkg-gencontrol`) if it has the same value as the `Version` field of the binary package in question. The field itself may be omitted from a binary package control file when the source package has the same name and version as the binary package.

D.2.6 Package interrelationship fields: `Depends`, `Pre-Depends`, `Recommends` `Suggests`, `Conflicts`, `Provides`, `Replaces`

These fields describe the package's relationships with other packages. Their syntax and semantics are described in 'Declaring relationships between packages' on page 37.

D.2.7 `Description`

In a binary package `Packages` file or main source control file this field contains a description of the binary package, in a special format. See 'Descriptions of packages - the `Description` field' on page 28 for

[3] It is usual to leave a space after the package name if a version number is specified.

details.

In a `.changes` file it contains a summary of the descriptions for the packages being uploaded. The part of the field before the first newline is empty; thereafter each line has the name of a binary package and the summary description line from that binary package. Each line is indented by one space.

D.2.8 `Essential`

This is a boolean field which may occur only in the control file of a binary package (or in the `Packages` file) or in a per-package fields paragraph of a main source control data file.

If set to `yes` then `dpkg` and `dselect` will refuse to remove the package (though it can be upgraded and/or replaced). The other possible value is `no`, which is the same as not having the field at all.

D.2.9 `Section` and `Priority`

These two fields classify the package. The `Priority` represents how important that it is that the user have it installed; the `Section` represents an application area into which the package has been classified.

When they appear in the `debian/control` file these fields give values for the section and priority subfields of the `Files` field of the `.changes` file, and give defaults for the section and priority of the binary packages.

The section and priority are represented, though not as separate fields, in the information for each file in the `-File`field of a `.changes` file. The section value in a `.changes` file is used to decide where to install a package in the FTP archive.

These fields are not used by by `dpkg` proper, but by `dselect` when it sorts packages and selects defaults. See the Debian policy manual for the priorities in use and the criteria for selecting the priority for a Debian package, and look at the Debian FTP archive for a list of currently in-use priorities.

These fields may appear in binary package control files, in which case they provide a default value in case the `Packages` files are missing the information. `dpkg` and `dselect` will only use the value from a `.deb` file if they have no other information; a value listed in a `Packages` file will always take precedence. By default `dpkg-gencontrol` does not include the section and priority in the control file of a binary package - use the `-isp`, `-is` or `-ip` options to achieve this effect.

D.2.10 `Binary`

This field is a list of binary packages.

When it appears in the `.dsc` file it is the list of binary packages which a source package can produce. It does not necessarily produce all of these binary packages for every architecture. The source control file doesn't contain details of which architectures are appropriate for which of the binary packages.

When it appears in a `.changes` file it lists the names of the binary packages actually being uploaded.

The syntax is a list of binary packages separated by commas. [4] Currently the packages must be separated using only spaces in the `.changes` file.

D.2.11 `Installed-Size`

This field appears in the control files of binary packages, and in the `Packages` files. It gives the total amount of disk space required to install the named package.

The disk space is represented in kilobytes as a simple decimal number.

D.2.12 `Files`

This field contains a list of files with information about each one. The exact information and syntax varies with the context. In all cases the the part of the field contents on the same line as the field name is empty. The remainder of the field is one line per file, each line being indented by one space and containing a number of sub-fields separated by spaces.

In the `.dsc` (Debian source control) file each line contains the MD5 checksum, size and filename of the tarfile and (if applicable) diff file which make up the remainder of the source package. [5] The exact forms of the filenames are described in 'Source packages as archives' on page 112.

In the `.changes` file this contains one line per file being uploaded. Each line contains the MD5 checksum, size, section and priority and the filename. The section and priority are the values of the corresponding fields in the main source control file - see '`Section` and `Priority`' on the facing page. If no section or priority is specified then `-` should be used, though section and priority values must be specified for new packages to be installed properly.

The special value `byhand` for the section in a `.changes` file indicates that the file in question is not an ordinary package file and must by installed by hand by the distribution maintainers. If the section is `byhand` the priority should be `-`.

If a new Debian revision of a package is being shipped and no new original source archive is being distributed the `.dsc` must still contain the `Files` field entry for the original source archive *`package-upstream-version`*`.orig.tar.gz`, but the `.changes` file should leave it out. In this case the original source archive on the distribution site must match exactly, byte-for-byte, the original source archive which was used to generate the `.dsc` file and diff which are being uploaded.

[4] A space after each comma is conventional.

[5] That is, the parts which are not the `.dsc`.

D.2.13 `Standards-Version`

The most recent version of the standards (the `dpkg` programmers' and policy manuals and associated texts) with which the package complies. This is updated manually when editing the source package to conform to newer standards; it can sometimes be used to tell when a package needs attention.

Its format is the same as that of a version number except that no epoch or Debian revision is allowed - see 'Version numbering' on page 19.

D.2.14 `Distribution`

In a `.changes` file or parsed changelog output this contains the (space-separated) name(s) of the distribution(s) where this version of the package should be or was installed. Distribution names follow the rules for package names. (See '`Package`' on page 116).

Current distribution values are:

stable This is the current 'released' version of Debian GNU/Linux. A new version is released approximately every 3 months after the *development* code has been *frozen* for a month of testing. Once the distribution is *stable* only major bug fixes are allowed. When changes are made to this distribution, the release number is increased (for example: 1.2r1 becomes 1.2r2 then 1.2r3, etc).

unstable This distribution value refers to the *developmental* part of the Debian distribution tree. New packages, new upstream versions of packages and bug fixes go into the *unstable* directory tree. Download from this distribution at your own risk.

contrib The packages with this distribution value do not meet the criteria for inclusion in the main Debian distribution as defined by the Policy Manual, but meet the criteria for the *contrib* Distribution. There is currently no distinction between stable and unstable packages in the *contrib* or *non-free* distributions. Use your best judgement in downloading from this Distribution.

non-free Like the packages in the *contrib* seciton, the packages in *non-free* do not meet the criteria for inclusion in the main Debian distribution as defined by the Policy Manual. Again, use your best judgement in downloading from this Distribution.

experimental The packages with this distribution value are deemed by their maintainers to be high risk. Oftentimes they represent early beta or developmental packages from various sources that the maintainers want people to try, but are not ready to be a part of the other parts of the Debian distribution tree. Download at your own risk.

frozen From time to time, (currently, every 3 months) the *unstable* distribution enters a state of 'code-freeze' in anticipation of release as a *stable* version. During this period of testing (usually 4 weeks) only fixes for existing or newly-discovered bugs will be allowed.

You should list *all* distributions that the package should be installed into. Except in unusual circumstances, installations to *stable* should also go into *frozen* (if it exists) and *unstable*. Likewise, installations into *frozen* should also go into *unstable*.

D.2.15 `Urgency`

This is a description of how important it is to upgrade to this version from previous ones. It consists of a single keyword usually taking one of the values `LOW`, `MEDIUM` or `HIGH`) followed by an optional commentary (separated by a space) which is usually in parentheses. For example:

```
Urgency: LOW (HIGH for diversions users)
```

This field appears in the `.changes` file and in parsed changelogs; its value appears as the value of the `urgency` attribute in a `dpkg`-style changelog (see ‘`debian/changelog`’ on page 109).

Urgency keywords are not case-sensitive.

D.2.16 `Date`

In `.changes` files and parsed changelogs, this gives the date the package was built or last edited.

D.2.17 `Format`

This field occurs in `.changes` files, and specifies a format revision for the file. The format described here is version `1.5`. The syntax of the format value is the same as that of a package version number except that no epoch or Debian revision is allowed - see ‘Version numbering’ on page 19.

D.2.18 `Changes`

In a `.changes` file or parsed changelog this field contains the human-readable changes data, describing the differences between the last version and the current one.

There should be nothing in this field before the first newline; all the subsequent lines must be indented by at least one space; blank lines must be represented by a line consiting only of a space and a full stop.

Each version's change information should be preceded by a ‘title’ line giving at least the version, distribution(s) and urgency, in a human-readable way.

If data from several versions is being returned the entry for the most recent version should be returned first, and entries should be separated by the representation of a blank line (the ‘title’ line may also be followed by the representation of blank line).

D.2.19 `Filename` and `MSDOS-Filename`

These fields in `Packages` files give the filename(s) of (the parts of) a package in the distribution directories, relative to the root of the Debian hierarchy. If the package has been split into several parts the parts are all listed in order, separated by spaces.

D.2.20 `Size` and `MD5sum`

These fields in `Packages` files give the size (in bytes, expressed in decimal) and MD5 checksum of the file(s) which make(s) up a binary package in the distribution. If the package is split into several parts the values for the parts are listed in order, separated by spaces.

D.2.21 `Status`

This field in `dpkg`'s status file records whether the user wants a package installed, removed or left alone, whether it is broken (requiring reinstallation) or not and what its current state on the system is. Each of these pieces of information is a single word.

D.2.22 `Config-Version`

If a package is not installed or not configured, this field in `dpkg`'s status file records the last version of the package which was successfully configured.

D.2.23 `Conffiles`

This field in `dpkg`'s status file contains information about the automatically-managed configuration files held by a package. This field should *not* appear anywhere in a package!

D.2.24 Obsolete fields

These are still recognised by `dpkg` but should not appear anywhere any more.

`Revision`

`Package-Revision`

`Package_Revision` The Debian revision part of the package version was at one point in a separate control file field. This field went through several names.

`Recommended` Old name for `Recommends`

`Optional` Old name for `Suggests`.

`Class` Old name for `Priority`.

Appendix E

Configuration file handling (from old Packaging Manual)

`dpkg` can do a certain amount of automatic handling of package configuration files.

Whether this mechanism is appropriate depends on a number of factors, but basically there are two approaches to any particular configuration file.

The easy method is to ship a best-effort configuration in the package, and use `dpkg`'s conffile mechanism to handle updates. If the user is unlikely to want to edit the file, but you need them to be able to without losing their changes, and a new package with a changed version of the file is only released infrequently, this is a good approach.

The hard method is to build the configuration file from scratch in the `postinst` script, and to take the responsibility for fixing any mistakes made in earlier versions of the package automatically. This will be appropriate if the file is likely to need to be different on each system.

E.1 Automatic handling of configuration files by `dpkg`

A package may contain a control area file called `conffiles`. This file should be a list of filenames of configuration files needing automatic handling, separated by newlines. The filenames should be absolute pathnames, and the files referred to should actually exist in the package.

When a package is upgraded `dpkg` will process the configuration files during the configuration stage, shortly before it runs the package's `postinst` script,

For each file it checks to see whether the version of the file included in the package is the same as the one that was included in the last version of the package (the one that is being upgraded from); it also compares the version currently installed on the system with the one shipped with the last version.

If neither the user nor the package maintainer has changed the file, it is left alone. If one or the other has changed their version, then the changed version is preferred - i.e., if the user edits their file, but the package maintainer doesn't ship a different version, the user's changes will stay, silently, but if the maintainer ships a new version and the user hasn't edited it the new version will be installed (with an informative message). If both have changed their version the user is prompted about the problem and must resolve the differences themselves.

The comparisons are done by calculating the MD5 message digests of the files, and storing the MD5 of the file as it was included in the most recent version of the package.

When a package is installed for the first time `dpkg` will install the file that comes with it, unless that would mean overwriting a file already on the filesystem.

However, note that `dpkg` will *not* replace a conffile that was removed by the user (or by a script). This is necessary because with some programs a missing file produces an effect hard or impossible to achieve in another way, so that a missing file needs to be kept that way if the user did it.

Note that a package should *not* modify a `dpkg`-handled conffile in its maintainer scripts. Doing this will lead to `dpkg` giving the user confusing and possibly dangerous options for conffile update when the package is upgraded.

E.2 Fully-featured maintainer script configuration handling

For files which contain site-specific information such as the hostname and networking details and so forth, it is better to create the file in the package's `postinst` script.

This will typically involve examining the state of the rest of the system to determine values and other information, and may involve prompting the user for some information which can't be obtained some other way.

When using this method there are a couple of important issues which should be considered:

If you discover a bug in the program which generates the configuration file, or if the format of the file changes from one version to the next, you will have to arrange for the postinst script to do something sensible - usually this will mean editing the installed configuration file to remove the problem or change the syntax. You will have to do this very carefully, since the user may have changed the file, perhaps to fix the very problem that your script is trying to deal with - you will have to detect these situations and deal with them correctly.

If you do go down this route it's probably a good idea to make the program that generates the configuration file(s) a separate program in `/usr/sbin`, by convention called *package*`config` and then run that if appropriate from the post-installation script. The *package*`config` program should not unquestioningly overwrite an existing configuration - if its mode of operation is geared towards setting up a package for the first time (rather than any arbitrary reconfiguration later) you should have it check whether the configuration already exists, and require a `--force` flag to overwrite it.

Appendix F

Alternative versions of an interface - `update-alternatives` (from old Packaging Manual)

When several packages all provide different versions of the same program or file it is useful to have the system select a default, but to allow the system administrator to change it and have their decisions respected.

For example, there are several versions of the `vi` editor, and there is no reason to prevent all of them from being installed at once, each under their own name (`nvi`, `vim` or whatever). Nevertheless it is desirable to have the name `vi` refer to something, at least by default.

If all the packages involved cooperate, this can be done with `update-alternatives`.

Each package provides its own version under its own name, and calls `update-alternatives` in its postinst to register its version (and again in its prerm to deregister it).

See the manpage `update-alternatives(8)` for details.

If `update-alternatives` does not seem appropriate you may wish to consider using diversions instead.

Appendix G

Diversions - overriding a package's version of a file (from old Packaging Manual)

It is possible to have `dpkg` not overwrite a file when it reinstalls the package it belongs to, and to have it put the file from the package somewhere else instead.

This can be used locally to override a package's version of a file, or by one package to override another's version (or provide a wrapper for it).

Before deciding to use a diversion, read 'Alternative versions of an interface - `update-alternatives` (from old Packaging Manual)' on page 125 to see if you really want a diversion rather than several alternative versions of a program.

There is a diversion list, which is read by `dpkg`, and updated by a special program `dpkg-divert`. Please see `dpkg-divert(8)` for full details of its operation.

When a package wishes to divert a file from another, it should call `dpkg-divert` in its preinst to add the diversion and rename the existing file. For example, supposing that a `smailwrapper` package wishes to install a wrapper around `/usr/sbin/smail`:

```
if [ install = "$1" -o upgrade = "$1" ]; then
   dpkg-divert --package smailwrapper --add --rename \
      --divert /usr/sbin/smail.real /usr/sbin/smail
fi
```

Testing `$1` is necessary so that the script doesn't try to add the diversion again when `smailwrapper` is upgraded. The `--package smailwrapper` ensures that `smailwrapper`'s copy of `/usr/sbin/smail` can bypass the diversion and get installed as the true version.

The postrm has to do the reverse:

```
if [ remove = "$1" ]; then
   dpkg-divert --package smailwrapper --remove --rename \
      --divert /usr/sbin/smail.real /usr/sbin/smail
fi
```

Do not attempt to divert a file which is vitally important for the system's operation - when using `dpkg-divert` there is a time, after it has been diverted but before `dpkg` has installed the new version, when the file does not exist.