Debian Policy Manual

The Debian Policy Mailing List

version 3.9.5.0, 2013-10-28

Abstract

This manual describes the policy requirements for the Debian 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.

Contents

Chapter 1

About this manual

1.1 Scope

This manual describes the policy requirements for the Debian 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.

The appendices to this manual are not necessarily normative, either. Please see 'Introduction and scope of these appendices' on page 91 for more information.

In the normative part of 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 *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.

udebs (stripped-down binary packages used by the Debian Installer) do not comply with all of the requirements discussed here. See the Debian Installer internals manual (`http://d-i.alioth.debian.org/doc/internals/ch03.html`) for more information about them.

1.2 New versions of this document

This manual is distributed via the Debian package `debian-policy` (`http://packages.debian.org/debian-policy`) (`packages.debian.org /debian-policy`).

The current version of this document is also available from the Debian web mirrors at `/doc/debian-policy/` (`http://www.debian.org/doc/debian-policy/`). (`www.debian.org /doc/debian-policy/`) Also available from the same directory are several other formats: `policy.html.tar.gz` (`/doc/debian-policy/policy.html.tar.gz`),

[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 interface not changing, and the package management software authors need to ensure compatibility with this interface definition. (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.

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

policy.pdf.gz (/doc/debian-policy/policy.pdf.gz) and policy.ps.gz (/doc/debian-policy/policy.ps.gz).

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

1.3 Authors and Maintainers

Originally called "Debian GNU/Linux Policy Manual", this manual was initially written in 1996 by Ian Jackson. It was revised on November 27th, 1996 by David A. Morris. Christian Schwarz added new sections on March 15th, 1997, and reworked/restructured it in April-July 1997. Christoph Lameter contributed the "Web Standard". Julian Gilbey largely restructured it in 2001.

Since September 1998, the responsibility for the contents of this document lies on the debian-policy mailing list (mailto:debian-policy@lists.debian.org). Proposals are discussed there and inserted into policy after a certain consensus is established. The actual editing is done by a group of maintainers that have no editorial powers. These are the current maintainers:

1 Russ Allbery

2 Bill Allombert

3 Andrew McMillan

4 Manoj Srivastava

5 Colin Watson

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.

Please do not try to reach the individual authors or maintainers of the Policy Manual regarding changes to the Policy.

1.4 Related documents

There are several other documents other than this Policy Manual that are necessary to fully understand some Debian policies and procedures.

The external "sub-policy" documents are referred to in:
- 'File System Structure' on page 59
- 'Virtual packages' on page 11
- 'Menus' on page 67
- 'Perl programs and modules' on page 84
- 'Prompting in maintainer scripts' on page 12
- 'Emacs lisp programs' on page 84

In addition to those, which carry the weight of policy, there is the Debian Developer's Reference. This document describes procedures and resources for Debian developers, but it is *not* normative; rather, it includes things that don't belong in the Policy, such as best practices for developers.

The Developer's Reference is available in the developers-reference package. It's also available from the Debian web mirrors at /doc/developers-reference/ (http://www.debian.org/doc/developers-reference/).

Finally, a specification for machine-readable copyright files is maintained as part of the debian-policy package using the same procedure as the other policy documents. Use of this format is optional.

1.5 Definitions

The following terms are used in this Policy Manual:

ASCII The character encoding specified by ANSI X3.4-1986 and its predecessor standards, referred to in MIME as US-ASCII, and corresponding to an encoding in eight bits per character of the first 128 Unicode (`http://www.unicode.org/`) characters, with the eighth bit always zero.

UTF-8 The transformation format (sometimes called encoding) of Unicode (`http://www.unicode.org/`) defined by RFC 3629 (`http://www.rfc-editor.org/rfc/rfc3629.txt`). UTF-8 has the useful property of having ASCII as a subset, so any text encoded in ASCII is trivially also valid UTF-8.

Chapter 2

The Debian Archive

The Debian system is maintained and distributed as a collection of *packages*. Since there are so many of them (currently well over 15000), 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 areas[1] based on their licenses and other restrictions.

The aims of this 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.

The *main* archive area forms the *Debian distribution*.

Packages in the other archive areas (`contrib`, `non-free`) 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 The Debian Free Software Guidelines

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

1. **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.

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

3. **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.

4. **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.)

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

6. **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.

[1] The Debian archive software uses the term “component” internally and in the Release file format to refer to the division of an archive. The Debian Social Contract simply refers to “areas.” This document uses terminology similar to the Social Contract.

7. 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.

8. 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.

9. 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.

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

2.2 Archive areas

2.2.1 The main archive area

The *main* archive area comprises the Debian distribution. Only the packages in this area are considered part of the distribution. None of the packages in the *main* archive area require software outside of that area to function. Anyone may use, share, modify and redistribute the packages in this archive area freely[2].

Every package in *main* must comply with the DFSG (Debian Free Software Guidelines).

In addition, the packages in *main*

- must not require or recommend a package outside of *main* for compilation or execution (thus, the package must not declare a "Pre-Depends", "Depends", "Recommends", "Build-Depends", or "Build-Depends-Indep" 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.

2.2.2 The contrib archive area

The *contrib* archive area contains supplemental packages intended to work with the Debian distribution, but which require software outside of the distribution to either build or function.

Every package in *contrib* must comply with the DFSG.

In addition, the packages in *contrib*

- must not be so buggy that we refuse to support them, and
- must meet all policy requirements presented in this manual.

Examples of packages which would be included in *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.

2.2.3 The non-free archive area

The *non-free* archive area contains supplemental packages intended to work with the Debian distribution that do not comply with the DFSG or have other problems that make their distribution problematic. They may not comply with all of the policy requirements in this manual due to restrictions on modifications or other limitations.

Packages must be placed in *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*

- 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. [3]

[2]See What Does Free Mean? (`http://www.debian.org/intro/free`) for more about what we mean by free software.

[3]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.

2.3 Copyright considerations

Every package must be accompanied by a verbatim copy of its copyright information and distribution license in the file `/usr/share/doc/`*`package`*`/copyright` (see 'Copyright information' on page 89 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".

2.4 Sections

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

The archive area and section for each package should be specified in the package's `Section` control record (see '`Section`' on page 26). 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:

- *section* if the package is in the *main* archive area,
- *area/section* if the package is in the *contrib* or *non-free* archive areas.

The Debian archive maintainers provide the authoritative list of sections. At present, they are: admin, cli-mono, comm, database, debug, devel, doc, editors, education, electronics, embedded, fonts, games, gnome, gnu-r, gnustep, graphics, hamradio, haskell, httpd, interpreters, introspection, java, kde, kernel, libdevel, libs, lisp, localization, mail, math, metapackages, misc, net, news, ocaml, oldlibs, otherosfs, perl, php, python, ruby, science, shells, sound, tasks, tex, text, utils, vcs, video, web, x11, xfce, zope. The additional section *debian-installer* contains special packages used by the installer and is not used for normal Debian packages.

For more information about the sections and their definitions, see the list of sections in unstable (`http://packages.debian.org/unstable/`).

2.5 Priorities

Each package should have a *priority* value, which is included in the package's *control record* (see '`Priority`' on page 26). This information is used by the Debian package management tools to separate high-priority packages from less-important packages.

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

`required` Packages which are necessary for the proper functioning of the system (usually, this means that dpkg functionality depends on these packages). Removing a `required` package may cause your system to become totally broken and you may not even be able to use `dpkg` to put things back, so only do so if you know what you are doing. 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.[4] 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.

`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 specialized requirements (such as packages containing only detached debugging symbols).

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.

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

Chapter 3

Binary packages

The Debian 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.

A `.deb` package contains two sets of files: a set of files to install on the system when the package is installed, and a set of files that provide additional metadata about the package or which are executed when the package is installed or removed. This second set of files is called *control information files*. Among those files are the package maintainer scripts and `control`, the binary package control file that contains the control fields for the package. Other control information files include the `symbols` file or `shlibs` file used to store shared library dependency information and the `conffiles` file that lists the package's configuration files (described in 'Configuration files' on page 74).

There is unfortunately a collision of terminology here between control information files and files in the Debian control file format. Throughout this document, a *control file* refers to a file in the Debian control file format. These files are documented in 'Control files and their fields' on page 23. Only files referred to specifically as *control information files* are the files included in the control information file member of the `.deb` file format used by binary packages. Most control information files are not in the Debian control file format.

3.1 The package name

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

The package name is included in the control field `Package`, the format of which is described in '`Package`' on page 27. The package name is also included as a part of the file name of the `.deb` file.

3.2 The version of a package

Every package has a version number recorded in its `Version` control file field, described in '`Version`' on page 28.

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.

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

3.2.1 Version numbers based on dates

In general, Debian packages should use the same version numbers as the upstream sources. However, upstream version numbers based on some date formats (sometimes used for development or "snapshot" releases) will not be ordered correctly by the package management software. For example, `dpkg` will consider "96May01" to be greater than "96Dec24".

To prevent having to use epochs for every new upstream version, the date-based portion of any upstream version number should be given in a way that sorts correctly: four-digit year first, followed by a two-digit numeric month, followed by a two-digit numeric date, possibly with punctuation between the components.

Native Debian packages (i.e., packages which have been written especially for Debian) whose version numbers include dates should also follow these rules. If punctuation is desired between the date components, remember that hyphen (`-`) cannot be used in native package versions. Period (`.`) is normally a good choice.

3.3 The maintainer of a package

Every package must have a maintainer, except for orphaned packages as described below. 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 maintaining the Debian packaging files, evaluating and responding appropriately to reported bugs, uploading new versions of the package (either directly or through a sponsor), ensuring that the package is placed in the appropriate archive area and included in Debian releases as appropriate for the stability and utility of the package, and requesting removal of the package from the Debian distribution if it is no longer useful or maintainable.

The maintainer must be specified in the `Maintainer` control field with their correct name and a working email address. The email address given in the `Maintainer` control field must accept mail from those role accounts in Debian used to send automated mails regarding the package. This includes non-spam mail from the bug-tracking system, all mail from the Debian archive maintenance software, and other role accounts or automated processes that are commonly agreed on by the project.[1] If one person or team maintains several packages, they should use the same form of their name and email address in the `Maintainer` fields of those packages.

The format of the `Maintainer` control field is described in '`Maintainer`' on page 26.

If the maintainer of the package is a team of people with a shared email address, the `Uploaders` control field must be present and must contain at least one human with their personal email address. See '`Uploaders`' on page 26 for the syntax of that field.

An orphaned package is one with no current maintainer. Orphaned packages should have their `Maintainer` control field set to `Debian QA Group <packages@qa.debian.org>`. These packages are considered maintained by the Debian project as a whole until someone else volunteers to take over maintenance.[2]

3.4 The description of a package

Every Debian package must have a `Description` control field which contains a synopsis and extended description of the package. Technical information about the format of the `Description` field is in '`Description`' on page 29.

The description should describe the package (the program) to a user (system administrator) who has never met it before so that they have enough information to decide whether they want to install it. This description should not just be copied verbatim from the program's documentation.

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.

The description 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.

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).

3.4.1 The single line synopsis

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.

[1] A sample implementation of such a whitelist written for the Mailman mailing list management software is used for mailing lists hosted by alioth.debian.org.

[2] The detailed procedure for gracefully orphaning a package can be found in the Debian Developer's Reference (see 'Related documents' on page 2).

3.4.2 The extended description

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.[3]

3.5 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.[4]

Sometimes, unpacking one package requires that another package be first unpacked *and* configured. In this case, the depending package must specify this dependency in the `Pre-Depends` control field.

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.

The format of the package interrelationship control fields is described in 'Declaring relationships between packages' on page 41.

3.6 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. (See also 'Virtual packages - `Provides`' on page 45)

The latest version of the authoritative list of virtual package names can be found in the `debian-policy` package. It is also available from the Debian web mirrors at `/doc/packaging-manuals/virtual-package-names-list.txt` (`http://www.debian.org/doc/packaging-manuals/virtual-package-names-list.txt`).

The procedure for updating the list is described in the preface to the list.

3.7 Base system

The `base system` is a minimum subset of the Debian system that is installed before everything else on a new system. Only very few packages are allowed to form part of the base system, in order to keep the required disk usage very small.

The base system consists of all those packages with priority `required` or `important`. Many of them will be tagged `essential` (see below).

[3]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.

[4]Essential is needed in part to avoid unresolvable dependency loops on upgrade. If packages add unnecessary dependencies on packages in this set, the chances that there **will** be an unresolvable dependency loop caused by forcing these Essential packages to be configured first before they need to be is greatly increased. It also increases the chances that frontends will be unable to **calculate** an upgrade path, even if one exists. Also, functionality is rarely ever removed from the Essential set, but *packages* have been removed from the Essential set when the functionality moved to a different package. So depending on these packages *just in case* they stop being essential does way more harm than good.

3.8 Essential packages

Essential is defined as the minimal set of functionality that must be available and usable on the system at all times, even when packages are in the "Unpacked" state. Packages are tagged `essential` for a system using the `Essential` control field. The format of the `Essential` control field is described in '`Essential`' on page 27.

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.

Maintainers should take great care in adding any programs, interfaces, or functionality to `essential` packages. Packages may assume that functionality provided by `essential` packages is always available without declaring explicit dependencies, which means that removing functionality from the Essential set is very difficult and is almost never done. Any capability added to an `essential` package therefore creates an obligation to support that capability as part of the Essential set in perpetuity.

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.

3.9 Maintainer Scripts

The package installation scripts should avoid producing output which 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, not passing the `--verbose` option to `update-alternatives`.

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 73 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. When adding or removing diversions, package maintainer scripts must provide the `--package` flag to `dpkg-divert` and must not use `--local`.

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 removed. (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.)

3.9.1 Prompting in maintainer scripts

Package maintainer scripts may prompt the user if necessary. Prompting must be done by communicating through a program, such as `debconf`, which conforms to the Debian Configuration Management Specification, version 2 or higher.

Packages which are essential, or which are dependencies of essential packages, may fall back on another prompting method if no such interface is available when they are executed.

The Debian Configuration Management Specification is included in the `debconf_specification` files in the `debian-policy` package. It is also available from the Debian web mirrors at `/doc/packaging-manuals/debconf_specification.html` (`http://www.debian.org/doc/packaging-manuals/debconf_specification.html`).

Packages which use the Debian Configuration Management Specification may contain the additional control information files `config` and `templates`. `config` is an additional maintainer script used for package configuration, and `templates` contains templates used for user prompting. The `config` script might be run before the `preinst` script and before the package is unpacked or any of its dependencies or pre-dependencies are satisfied. Therefore it must work using only the tools present in *essential* packages.[5]

[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.

Packages which use the Debian Configuration Management Specification must allow for translation of their user-visible messages by using a gettext-based system such as the one provided by the `po-debconf` package.

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).

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`.

Chapter 4

Source packages

4.1 Standards conformance

Source packages should specify the most recent version number of this policy document with which your package complied when it was last updated.

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

The version is specified in the `Standards-Version` control field. The format of the `Standards-Version` field is described in '`Standards-Version`' on page 28.

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.[1]

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 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).[2]

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.[3]

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.

'Declaring relationships between packages' on page 41 explains the technical details.

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

[2]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 tasks 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.

[3]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.

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 without losing the changes you made.

4.4 Debian changelog: `debian/changelog`

Changes in the Debian version of the package should be briefly explained in the Debian changelog file `debian/changelog`.[4] This includes modifications made in the Debian package compared to the upstream one as well as other changes and updates to the package. [5]

The format of the `debian/changelog` 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
     [optional blank line(s), stripped]
  * change details
    more change details
     [blank line(s), included in output of dpkg-parsechangelog]
  * even more change details
     [optional blank line(s), stripped]
 -- 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 29.

urgency is the value for the `Urgency` field in the `.changes` file for the upload (see '`Urgency`' on page 30). 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.

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] This information is conveyed via the `Closes` field in the `.changes` file (see '`Closes`' on page 31).

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.[7] The information here will be copied to the `Changed-By`

[4] Mistakes in changelogs are usually best rectified by making a new changelog entry rather than "rewriting history" by editing old changelog entries.

[5] Although there is nothing stopping an author who is also the Debian maintainer from using this changelog 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.

[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 software (`dak`) using the *version* of the changelog entry.

[7] If the developer uploading the package is not one of the usual maintainers of the package (as listed in the `Maintainer` or `Uploaders` control fields of the package), the first line of the changelog is conventionally used to explain why a non-maintainer is uploading the package. The Debian Developer's Reference (see 'Related documents' on page 2) documents the conventions used.

field in the `.changes` file (see 'Changed-By' on page 26), and then later used to send an acknowledgement when the upload has been installed.

The *date* has the following format[8] (compatible and with the same semantics of RFC 2822 and RFC 5322):

```
day-of-week, dd month yyyy hh:mm:ss +zzzz
```

where:

- day-of week is one of: Mon, Tue, Wed, Thu, Fri, Sat, Sun
- dd is a one- or two-digit day of the month (01-31)
- month is one of: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
- yyyy is the four-digit year (e.g. 2010)
- hh is the two-digit hour (00-23)
- mm is the two-digit minutes (00-59)
- ss is the two-digit seconds (00-60)
- +zzzz or -zzzz is the the time zone offset from Coordinated Universal Time (UTC). "+" indicates that the time is ahead of (i.e., east of) UTC and "-" indicates that the time is behind (i.e., west of) UTC. The first two digits indicate the hour difference from UTC and the last two digits indicate the number of additional minutes difference from UTC. The last two digits must be in the range 00-59.

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

The entire changelog must be encoded in UTF-8.

For more information on placement of the changelog files within binary packages, please see 'Changelog files' on page 89.

4.5 Copyright: `debian/copyright`

Every package must be accompanied by a verbatim copy of its copyright information and distribution license in the file `/usr/share/doc/`*`package`*`/copyright` (see 'Copyright information' on page 89 for further details). Also see 'Copyright considerations' on page 7 for further considerations related to copyrights for packages.

4.6 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.

4.7 Time Stamps

Maintainers should preserve the modification times of the upstream source files in a package, as far as is reasonably possible.[9]

[8] This is the same as the format generated by `date -R`.

[9] 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.

4.8 Restrictions on objects in source packages

The source package may not contain any hard links[10], device special files, sockets or setuid or setgid files.[11]

4.9 Main building script: `debian/rules`

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. That is, invoking either of `make -f debian/rules` *`args`*`...` or `./debian/rules` *`args`*`...` must result in identical behavior.

The following targets are required and must be implemented by `debian/rules`: `clean`, `binary`, `binary-arch`, `binary-indep`, `build`, `build-arch` and `build-indep`. These are the targets called by `dpkg-buildpackage`.

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. It also follows that any target that these targets depend on must also be non-interactive.

The targets are as follows:

`build` (required) The `build` target should perform all the configuration and compilation of the package. If a package has an interactive pre-build configuration routine, the Debian source package must either be built after this has taken place (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.[12]

`build-arch` (required), `build-indep` (required) The `build-arch` target must perform all the 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 must perform all the 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 either depend on those targets or take the same actions as invoking those targets would perform.[13]

The `build-arch` and `build-indep` targets must not do anything that might require root privilege.

`binary` (required), `binary-arch` (required), `binary-indep` (required) The `binary` target must be all that is necessary for the user to build the binary package(s) produced from this source package. 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

[10]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.

[11]Setgid directories are allowed.

[12]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.

[13]This split allows binary-only builds to not install the dependencies required for the `build-indep` target and skip any resource-intensive build tasks that are only required when building architecture-independent binary packages.

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.[14]

clean (required) 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.

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.

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.

patch (optional) This target performs whatever additional actions are required to make the source ready for editing (unpacking additional upstream archives, applying patches, etc.). It is recommended to be implemented for any package where dpkg-source -x does not result in source ready for additional modification. See 'Source package handling: debian/README.source' on page 21.

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 architecture as well as for the host architecture. The build architecture is the architecture on which debian/rules is run and the package build is performed. The host architecture is the architecture on which the resulting package will be installed and run. These are normally the same, but may be different in the case of cross-compilation (building packages for one architecture on machines of a different architecture).

Here is a list of supported make variables:
- DEB_*_ARCH (the Debian architecture)
- DEB_*_ARCH_CPU (the Debian CPU name)
- DEB_*_ARCH_OS (the Debian System name)
- 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 architecture or HOST for specification of the host architecture.

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 DEB_*_ARCH_CPU and DEB_*_ARCH_OS variables should be used for that. GNU style variables should generally only be used with upstream build systems.

4.9.1 debian/rules and DEB_BUILD_OPTIONS

Supporting the standardized environment variable DEB_BUILD_OPTIONS is recommended. This variable can contain several flags to change how a package is compiled and built. Each flag must be in the form *flag* or *flag=options*. If multiple flags are given, they must be separated by whitespace.[15] *flag* must start with a lowercase letter (a-z) and consist only of

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

[15] Some packages support any delimiter, but whitespace is the easiest to parse inside a makefile and avoids ambiguity with flag values that contain commas.

lowercase letters, numbers (0-9), and the characters - and _ (hyphen and underscore). *options* must not contain whitespace. The same tag should not be given multiple times with conflicting values. Package maintainers may assume that DEB_BUILD_OPTIONS will not contain conflicting tags.

The meaning of the following tags has been standardized:

nocheck This tag says to not run any build-time test suite provided by the package.

noopt The presence of this tag means that the package should be compiled with a minimum of optimization. For C programs, it is best to add -O0 to CFLAGS (although this is usually the default). Some programs might fail to build or run at this level of optimization; it may be necessary to use -O1, for example.

nostrip This tag means that the debugging symbols should not be stripped from the binary during installation, so that debugging information may be included in the package.

parallel=n This tag means that the package should be built using up to n parallel processes if the package build system supports this.[16] If the package build system does not support parallel builds, this string must be ignored. If the package build system only supports a lower level of concurrency than *n*, the package should be built using as many parallel processes as the package build system supports. It is up to the package maintainer to decide whether the package build times are long enough and the package build system is robust enough to make supporting parallel builds worthwhile.

Unknown flags must be ignored by debian/rules.

The following makefile snippet is an example of how one may implement the build options; you will probably have to massage this example in order to make it work for your package.

```
CFLAGS = -Wall -g
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 (,$(filter noopt,$(DEB_BUILD_OPTIONS)))
    CFLAGS += -O0
else
    CFLAGS += -O2
endif
ifeq (,$(filter nostrip,$(DEB_BUILD_OPTIONS)))
    INSTALL_PROGRAM += -s
endif
ifneq (,$(filter parallel=%,$(DEB_BUILD_OPTIONS)))
    NUMJOBS = $(patsubst parallel=%,%,$(filter parallel=%,$(DEB_BUILD_OPTIONS)))
    MAKEFLAGS += -j$(NUMJOBS)
endif

build:
 # ...
ifeq (,$(filter nocheck,$(DEB_BUILD_OPTIONS)))
 # Code to run the package test suite.
endif
```

4.10 Variable substitutions: debian/substvars

When dpkg-gencontrol generates binary package control files (DEBIAN/control), it performs variable substitutions on its 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 which case it must be removed by the clean target.

See deb-substvars(5) for full details about source variable substitutions, including the format of debian/substvars.

4.11 Optional upstream source location: debian/watch

This is an optional, recommended configuration file for the uscan utility which defines how to automatically scan ftp or http sites for newly available updates of the package. This is used by http://dehs.alioth.debian.org/ and other Debian QA tools to help with quality control and maintenance of the distribution as a whole.

[16] Packages built with make can often implement this by passing the -j*n* option to make.

4.12 Generated files list: `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`[17]) 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.

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`.

4.13 Convenience copies of code

Some software packages include in their distribution convenience copies of code from other software packages, generally so that users compiling from source don't have to download multiple packages. Debian packages should not make use of these convenience copies unless the included package is explicitly intended to be used in this way.[18] If the included code is already in the Debian archive in the form of a library, the Debian packaging should ensure that binary packages reference the libraries already in Debian and the convenience copy is not used. If the included code is not already in Debian, it should be packaged separately as a prerequisite if possible. [19]

4.14 Source package handling: `debian/README.source`

If running `dpkg-source -x` on a source package doesn't produce the source of the package, ready for editing, and allow one to make changes and run `dpkg-buildpackage` to produce a modified package without taking any additional steps, creating a `debian/README.source` documentation file is recommended. This file should explain how to do all of the following:

1. Generate the fully patched source, in a form ready for editing, that would be built to create Debian packages. Doing this with a `patch` target in `debian/rules` is recommended; see 'Main building script: `debian/rules`' on page 18.
2. Modify the source and save those modifications so that they will be applied when building the package.
3. Remove source modifications that are currently being applied when building the package.
4. Optionally, document what steps are necessary to upgrade the Debian source package to a new upstream version, if applicable.

This explanation should include specific commands and mention any additional required Debian packages. It should not assume familiarity with any specific Debian packaging system or patch management tools.

This explanation may refer to a documentation file installed by one of the package's build dependencies provided that the referenced documentation clearly explains these tasks and is not a general reference manual.

`debian/README.source` may also include any other information that would be helpful to someone modifying the source package. Even if the package doesn't fit the above description, maintainers are encouraged to document in a `debian/README.source` file any source package with a particularly complex or unintuitive source layout or build system (for example, a package that builds the same source multiple times to generate different binary packages).

[17] `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.

[18] For example, parts of the GNU build system work like this.

[19] Having multiple copies of the same code in Debian is inefficient, often creates either static linking or shared library conflicts, and, most importantly, increases the difficulty of handling security vulnerabilities in the duplicated code.

Chapter 5

Control files and their fields

The package management system manipulates data represented in a common format, known as *control data*, stored in *control files*. Control files are used for source packages, binary packages and the `.changes` files which control the installation of uploaded files[1].

5.1 Syntax of control files

A control file consists of one or more paragraphs of fields[2]. The paragraphs are separated by empty lines. Parsers may accept lines consisting solely of spaces and tabs as paragraph separators, but control files should use empty 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.) The ordering of the paragraphs in control files is significant.

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. The field name is composed of US-ASCII characters excluding control characters, space, and colon (i.e., characters in the ranges 33-57 and 59-126, inclusive). Field names must not begin with the comment character, `#`, nor with the hyphen character, `-`.

The field ends at the end of the line or at the end of the last continuation line (see below). 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`.

A paragraph must not contain more than one instance of a particular field name.

There are three types of fields:

simple The field, including its value, must be a single line. Folding of the field is not permitted. This is the default field type if the definition of the field does not specify a different type.

folded The value of a folded field is a logical line that may span several lines. The lines after the first are called continuation lines and must start with a space or a tab. Whitespace, including any newlines, is not significant in the field values of folded fields.[3]

multiline The value of a multiline field may comprise multiple continuation lines. The first line of the value, the part on the same line as the field name, often has special significance or may have to be empty. Other lines are added following the same syntax as the continuation lines of the folded fields. Whitespace, including newlines, is significant in the values of multiline fields.

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.

[1] `dpkg`'s internal databases are in a similar format.

[2] The paragraphs are also sometimes referred to as stanzas.

[3] This folding method is similar to RFC 5322, allowing control files that contain only one paragraph and no multiline fields to be read by parsers written for RFC 5322.

The presence and purpose of a field, and the syntax of its value may differ between types of control files.

Field names are not case-sensitive, but it is usual to capitalize the field names using mixed case as shown below. Field values are case-sensitive unless the description of the field says otherwise.

Paragraph separators (empty lines) and lines consisting only of spaces and tabs are not allowed within field values or between fields. Empty lines in field values are usually escaped by representing them by a space followed by a dot.

Lines starting with # without any preceding whitespace are comments lines that are only permitted in source package control files (`debian/control`). These comment lines are ignored, even between two continuation lines. They do not end logical lines.

All control files must be encoded in UTF-8.

5.2 Source package control files – `debian/control`

The `debian/control` file contains the most vital (and version-independent) information about the source package and about the binary packages it creates.

The first paragraph of the control file contains information about the source package in general. The subsequent sets each describe a binary package that the source tree builds.

The fields in the general paragraph (the first one, for the source package) are:

- `Source` (mandatory)
- `Maintainer` (mandatory)
- `Uploaders`
- `Section` (recommended)
- `Priority` (recommended)
- `Build-Depends` et al
- `Standards-Version` (recommended)
- `Homepage`
- `Vcs-Browser`, `Vcs-Git`, et al.

The fields in the binary package paragraphs are:

- `Package` (mandatory)
- `Architecture` (mandatory)
- `Section` (recommended)
- `Priority` (recommended)
- `Essential`
- `Depends` et al
- `Description` (mandatory)
- `Homepage`
- `Built-Using`
- `Package-Type`

The syntax and semantics of the fields are described below.

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. Some fields are folded in `debian/control`, but not in any other control file. These tools are responsible for removing the line breaks from such fields when using fields from `debian/control` to generate other control files.

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 'Variable substitutions: `debian/substvars`' on page 20 for details.

5.3 Binary package control files – `DEBIAN/control`

The `DEBIAN/control` file contains the most vital (and version-dependent) information about a binary package. It consists of a single paragraph.

The fields in this file are:

- `Package` (mandatory)

- `Source`
- `Version` (mandatory)
- `Section` (recommended)
- `Priority` (recommended)
- `Architecture` (mandatory)
- `Essential`
- `Depends` et al
- `Installed-Size`
- `Maintainer` (mandatory)
- `Description` (mandatory)
- `Homepage`
- `Built-Using`

5.4 Debian source control files – `.dsc`

This file consists of a single paragraph, possibly surrounded by a PGP signature. The fields of that paragraph are listed below. Their syntax is described above, in 'Syntax of control files' on page 23.

- `Format` (mandatory)
- `Source` (mandatory)
- `Binary`
- `Architecture`
- `Version` (mandatory)
- `Maintainer` (mandatory)
- `Uploaders`
- `Homepage`
- `Vcs-Browser`, `Vcs-Git`, et al.
- `Dgit`
- `Standards-Version` (recommended)
- `Build-Depends` et al
- `Package-List` (recommended)
- `Checksums-Sha1` and `Checksums-Sha256` (mandatory)
- `Files` (mandatory)

The Debian source 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.

5.5 Debian changes files – `.changes`

The `.changes` files are used by the Debian archive maintenance software to process updates to packages. They consist of a single paragraph, possibly surrounded by a PGP signature. That paragraph contains information from the `debian/control` file and other data about the source package gathered via `debian/changelog` and `debian/rules`.

`.changes` files have a format version that is incremented whenever the documented fields or their meaning change. This document describes format 1.8.

The fields in this file are:

- `Format` (mandatory)
- `Date` (mandatory)
- `Source` (mandatory)
- `Binary` (mandatory)
- `Architecture` (mandatory)
- `Version` (mandatory)
- `Distribution` (mandatory)
- `Urgency` (recommended)
- `Maintainer` (mandatory)
- `Changed-By`
- `Description` (mandatory)
- `Closes`
- `Changes` (mandatory)

- `Checksums-Sha1` and `Checksums-Sha256` (mandatory)
- `Files` (mandatory)

5.6 List of fields

5.6.1 `Source`

This field identifies the source package name.

In `debian/control` or a `.dsc` file, this field must contain only the name of the source package.

In a binary package control file or a `.changes` file, the source package name may be followed by a version number in parentheses[4]. 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.

Package names (both source and binary, see '`Package`' on the next page) must consist only 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 start with an alphanumeric character.

5.6.2 `Maintainer`

The package maintainer's name and email address. The name must 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).

See 'The maintainer of a package' on page 10 for additional requirements and information about package maintainers.

5.6.3 `Uploaders`

List of the names and email addresses of co-maintainers of the package, if any. If the package has other maintainers besides the one named in the Maintainer field, their names and email addresses should be listed here. The format of each entry is the same as that of the Maintainer field, and multiple entries must be comma separated.

This is normally an optional field, but if the `Maintainer` control field names a group of people and a shared email address, the `Uploaders` field must be present and must contain at least one human with their personal email address.

The Uploaders field in `debian/control` can be folded.

5.6.4 `Changed-By`

The name and email address of the person who prepared this version of the package, usually a maintainer. The syntax is the same as for the Maintainer field.

5.6.5 `Section`

This field specifies an application area into which the package has been classified. See 'Sections' on page 7.

When it appears in the `debian/control` file, it gives the value for the subfield of the same name in the `Files` field of the `.changes` file. It also gives the default for the same field in the binary packages.

5.6.6 `Priority`

This field represents how important it is that the user have the package installed. See 'Priorities' on page 7.

When it appears in the `debian/control` file, it gives the value for the subfield of the same name in the `Files` field of the `.changes` file. It also gives the default for the same field in the binary packages.

[4]It is customary to leave a space after the package name if a version number is specified.

5.6.7 `Package`

The name of the binary package.

Binary package names must follow the same syntax and restrictions as source package names. See '`Source`' on the preceding page for the details.

5.6.8 `Architecture`

Depending on context and the control file used, the `Architecture` field can include the following sets of values:

- A unique single word identifying a Debian machine architecture as described in 'Architecture specification strings' on page 79.
- An architecture wildcard identifying a set of Debian machine architectures, see 'Architecture wildcards' on page 79. `any` matches all Debian machine architectures and is the most frequently used.
- `all`, which indicates an architecture-independent package.
- `source`, which indicates a source package.

In the main `debian/control` file in the source package, this field may contain the special value `all`, the special architecture wildcard `any`, or a list of specific and wildcard architectures separated by spaces. If `all` or `any` appears, that value must be the entire contents of the field. Most packages will use either `all` or `any`.

Specifying a specific list of architectures indicates that the source will build an architecture-dependent package only on architectures included in the list. Specifying a list of architecture wildcards indicates that the source will build an architecture-dependent package on only those architectures that match any of the specified architecture wildcards. Specifying a list of architectures or architecture wildcards other than `any` is for the minority of cases where a program is not portable or is not useful on some architectures. Where possible, the program should be made portable instead.

In the Debian source control file `.dsc`, this field contains a list of architectures and architecture wildcards separated by spaces. When the list contains the architecture wildcard `any`, the only other value allowed in the list is `all`.

The list may include (or consist solely of) the special value `all`. In other words, in `.dsc` files unlike the `debian/control`, `all` may occur in combination with specific architectures. The `Architecture` field in the Debian source control file `.dsc` is generally constructed from the `Architecture` fields in the `debian/control` in the source package.

Specifying only `any` indicates that the source package isn't dependent on any particular architecture and should compile fine on any one. The produced binary package(s) will be specific to whatever the current build architecture is.

Specifying only `all` indicates that the source package will only build architecture-independent packages.

Specifying `any all` indicates that the source package isn't dependent on any particular architecture. The set of produced binary packages will include at least one architecture-dependant package and one architecture-independent package.

Specifying a list of architectures or architecture wildcards indicates that the source will build an architecture-dependent package, and will only work correctly on the listed or matching architectures. If the source package also builds at least one architecture-independent package, `all` will also be included in the list.

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 also being uploaded, the special entry `source` is also present. `all` will be present if any architecture-independent packages are being uploaded. Architecture wildcards such as `any` must never occur in the `Architecture` field in the `.changes` file.

See 'Main building script: `debian/rules`' on page 18 for information on how to get the architecture for the build process.

5.6.9 `Essential`

This is a boolean field which may occur only in the control file of a binary package or in a per-package fields paragraph of a source package control file.

If set to `yes` then the package management system will refuse to remove the package (upgrading and replacing it is still possible). The other possible value is `no`, which is the same as not having the field at all.

5.6.10 Package interrelationship fields: `Depends`, `Pre-Depends`, `Recommends`, `Suggests`, `Breaks`, `Conflicts`, `Provides`, `Replaces`, `Enhances`

These fields describe the package's relationships with other packages. Their syntax and semantics are described in 'Declaring relationships between packages' on page 41.

5.6.11 `Standards-Version`

The most recent version of the standards (the policy manual and associated texts) with which the package complies.

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 all four components may be specified.[5]

5.6.12 `Version`

The version number of a package. The 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[6] and the characters `. + - : ~` (full stop, plus, hyphen, colon, tilde) 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.

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 `+ . ~` (plus, full stop, tilde) 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 a Debian package, where the Debian package source must always be identical to the pristine source 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* is equivalent to a *debian_revision* of `0`.

When comparing two version numbers, first the *epoch* of each are compared, then the *upstream_version* if *epoch* is equal, and then *debian_revision* if *upstream_version* is also equal. *epoch* is compared numerically. The *upstream_version* and *debian_revision* parts are compared by the package management system using the following algorithm:

[5] 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.

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

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 and so that a tilde sorts before anything, even the end of a part. For example, the following parts are in sorted order from earliest to latest: `~~`, `~~a`, `~`, the empty part, `a`.[7]

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.[8]

5.6.13 `Description`

In a source or binary control file, the `Description` field contains a description of the binary package, consisting of two parts, the synopsis or the short description, and the long description. It is a multiline field with the following format:

```
Description: <single line synopsis>
 <extended description over several lines>
```

The lines in the extended description can have these formats:

- Those starting with a single space are part of a paragraph. Successive lines of this form will be word-wrapped when displayed. The leading space will usually be stripped off. The line must contain at least one non-whitespace character.
- Those starting with two or more spaces. These will be displayed verbatim. If the display cannot be panned horizontally, the displaying program will line wrap them "hard" (i.e., without taking account of word breaks). If it can they will be allowed to trail off to the right. None, one or two initial spaces may be deleted, but the number of spaces deleted from each line will be the same (so that you can have indenting work correctly, for example). The line must contain at least one non-whitespace character.
- Those containing a single space followed by a single full stop character. These are rendered as blank lines. This is the *only* way to get a blank line[9].
- Those containing a space, a full stop and some more characters. These are for future expansion. Do not use them.

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

See 'The description of a package' on page 10 for further information on this.

In a `.changes` file, the `Description` field contains a summary of the descriptions for the packages being uploaded. For this case, the first line of the field value (the part on the same line as `Description:`) is always empty. It is a multiline field, with one line per package. Each line is indented by one space and contains the name of a binary package, a space, a hyphen (`-`), a space, and the short description line from that package.

5.6.14 `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 installed. Valid distributions are determined by the archive maintainers.[10] The Debian

[7] One common use of `~` is for upstream pre-releases. For example, `1.0~beta1~svn1245` sorts earlier than `1.0~beta1`, which sorts earlier than `1.0`.

[8] 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.

[9] Completely empty lines will not be rendered as blank lines. Instead, they will cause the parser to think you're starting a whole new record in the control file, and will therefore likely abort with an error.

[10] Example distribution names in the Debian archive used in `.changes` files are:

unstable This distribution value refers to the *developmental* part of the Debian distribution tree. Most new packages, new upstream versions of packages and bug fixes go into the *unstable* directory tree.

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.

Others are used for updating stable releases or for security uploads. More information is available in the Debian Developer's Reference, section "The Debian archive".

archive software only supports listing a single distribution. Migration of packages to other distributions is handled outside of the upload process.

5.6.15 `Date`

This field includes the date the package was built or last edited. It must be in the same format as the *date* in a `debian/changelog` entry.

The value of this field is usually extracted from the `debian/changelog` file - see 'Debian changelog: `debian/changelog`' on page 16).

5.6.16 `Format`

In `.changes` files, this field declares the format version of that file. The syntax of the field value is the same as that of a package version number except that no epoch or Debian revision is allowed. The format described in this document is `1.8`.

In `.dsc` Debian source control files, this field declares the format of the source package. The field value is used by programs acting on a source package to interpret the list of files in the source package and determine how to unpack it. The syntax of the field value is a numeric major revision, a period, a numeric minor revision, and then an optional subtype after whitespace, which if specified is an alphanumeric word in parentheses. The subtype is optional in the syntax but may be mandatory for particular source format revisions. [11]

5.6.17 `Urgency`

This is a description of how important it is to upgrade to this version from previous ones. It consists of a single keyword taking one of the values `low`, `medium`, `high`, `emergency`, or `critical`[12] (not case-sensitive) followed by an optional commentary (separated by a space) which is usually in parentheses. For example:

```
Urgency: low (HIGH for users of diversions)
```

The value of this field is usually extracted from the `debian/changelog` file - see 'Debian changelog: `debian/changelog`' on page 16.

5.6.18 `Changes`

This multiline field contains the human-readable changes data, describing the differences between the last version and the current one.

The first line of the field value (the part on the same line as `Changes:`) is always empty. The content of the field is expressed as continuation lines, with each line indented by at least one space. Blank lines must be represented by a line consisting only of a space and a full stop (`.`).

The value of this field is usually extracted from the `debian/changelog` file - see 'Debian changelog: `debian/changelog`' on page 16).

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 a blank line).

[11]The source formats currently supported by the Debian archive software are `1.0`, `3.0 (native)`, and `3.0 (quilt)`.

[12]Other urgency values are supported with configuration changes in the archive software but are not used in Debian. The urgency affects how quickly a package will be considered for inclusion into the `testing` distribution and gives an indication of the importance of any fixes included in the upload. `Emergency` and `critical` are treated as synonymous.

5.6.19 `Binary`

This folded field is a list of binary packages. Its syntax and meaning varies depending on the control file in which it appears.

When it appears in the `.dsc` file, it lists binary packages which a source package can produce, separated by commas[13]. The source package 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 being uploaded, separated by whitespace (not commas).

5.6.20 `Installed-Size`

This field appears in the control files of binary packages, and in the `Packages` files. It gives an estimate of the total amount of disk space required to install the named package. Actual installed size may vary based on block size, file system properties, or actions taken by package maintainer scripts.

The disk space is given as the integer value of the estimated installed size in bytes, divided by 1024 and rounded up.

5.6.21 `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, Files is a multiline field. The first line of the field value (the part on the same line as `Files:`) is always empty. The content of the field is expressed as continuation lines, one line per file. Each line must be indented by one space and contain a number of sub-fields, separated by spaces, as described below.

In the `.dsc` file, each line contains the MD5 checksum, size and filename of the tar file and (if applicable) diff file which make up the remainder of the source package[14]. For example:

```
Files:
 c6f698f19f2a2aa07dbb9bbda90a2754 571925 example_1.2.orig.tar.gz
 938512f08422f3509ff36f125f5873ba 6220 example_1.2-1.diff.gz
```

The exact forms of the filenames are described in 'Source packages as archives' on page 97.

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. For example:

```
Files:
 4c31ab7bfc40d3cf49d7811987390357 1428 text extra example_1.2-1.dsc
 c6f698f19f2a2aa07dbb9bbda90a2754 571925 text extra example_1.2.orig.tar.gz
 938512f08422f3509ff36f125f5873ba 6220 text extra example_1.2-1.diff.gz
 7c98fe853b3bbb47a00e5cd129b6cb56 703542 text extra example_1.2-1_i386.deb
```

The section and priority are the values of the corresponding fields in the main source control file. 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.

5.6.22 `Closes`

A space-separated list of bug report numbers that the upload governed by the .changes file closes.

[13] A space after each comma is conventional.

[14] That is, the parts which are not the `.dsc`.

5.6.23 **`Homepage`**

The URL of the web site for this package, preferably (when applicable) the site from which the original source can be obtained and any additional upstream documentation or information may be found. The content of this field is a simple URL without any surrounding characters such as <>.

5.6.24 **`Checksums-Sha1`** and **`Checksums-Sha256`**

These multiline fields contain a list of files with a checksum and size for each one. Both `Checksums-Sha1` and `Checksums-Sha256` have the same syntax and differ only in the checksum algorithm used: SHA-1 for `Checksums-Sha1` and SHA-256 for `Checksums-Sha256`.

`Checksums-Sha1` and `Checksums-Sha256` are multiline fields. The first line of the field value (the part on the same line as `Checksums-Sha1:` or `Checksums-Sha256:`) is always empty. The content of the field is expressed as continuation lines, one line per file. Each line consists of the checksum, a space, the file size, a space, and the file name. For example (from a `.changes` file):

```
Checksums-Sha1:
 1f418afaa01464e63cc1ee8a66a05f0848bd155c 1276 example_1.0-1.dsc
 a0ed1456fad61116f868b1855530dbe948e20f06 171602 example_1.0.orig.tar.gz
 5e86ecf0671e113b63388dac81dd8d00e00ef298 6137 example_1.0-1.debian.tar.gz
 71a0ff7da0faaf608481195f9cf30974b142c183 548402 example_1.0-1_i386.deb
Checksums-Sha256:
 ac9d57254f7e835bed299926fd51bf6f534597cc3fcc52db01c4bffedae81272 1276 example_1.0-1.dsc
 0d123be7f51e61c4bf15e5c492b484054be7e90f3081608a5517007bfb1fd128 171602 example_1.0.orig.tar.gz
 f54ae966a5f580571ae7d9ef5e1df0bd42d63e27cb505b27957351a495bc6288 6137 example_1.0-1.debian.tar.gz
 3bec05c03974fdecd11d020fc2e8250de8404867a8a2ce865160c250eb723664 548402 example_1.0-1_i386.deb
```

In the `.dsc` file, these fields list all files that make up the source package. In the `.changes` file, these fields list all files being uploaded. The list of files in these fields must match the list of files in the `Files` field.

5.6.25 **`DM-Upload-Allowed`**

Obsolete, see below.

5.6.26 Version Control System (VCS) fields

Debian source packages are increasingly developed using VCSs. The purpose of the following fields is to indicate a publicly accessible repository where the Debian source package is developed.

`Vcs-Browser` URL of a web interface for browsing the repository.

`Vcs-Arch`, `Vcs-Bzr` (Bazaar), `Vcs-Cvs`, `Vcs-Darcs`, `Vcs-Git`, `Vcs-Hg` (Mercurial), `Vcs-Mtn` (Monotone), `Vcs-Svn` (Subver

The field name identifies the VCS. The field's value uses the version control system's conventional syntax for describing repository locations and should be sufficient to locate the repository used for packaging. Ideally, it also locates the branch used for development of new versions of the Debian package.

In the case of Git, the value consists of a URL, optionally followed by the word `-b` and the name of a branch in the indicated repository, following the syntax of the `git clone` command. If no branch is specified, the packaging should be on the default branch.

More than one different VCS may be specified for the same package.

5.6.27 **`Package-List`**

Multiline field listing all the packages that can be built from the source package, considering every architecture. The first line of the field value is empty. Each one of the next lines describes one binary package, by listing its name, type, section and priority separated by spaces. Fifth and subsequent space-separated items may be present and parsers must allow them. See the Package-Type field for a list of package types.

5.6.28 `Package-Type`

Simple field containing a word indicating the type of package: `deb` for binary packages and `udeb` for micro binary packages. Other types not defined here may be indicated. In source package control files, the `Package-Type` field should be omitted instead of giving it a value of `deb`, as this value is assumed for paragraphs lacking this field.

5.6.29 `Dgit`

Folded field containing a single git commit hash, presented in full, followed optionally by whitespace and other data to be defined in future extensions.

Declares that the source package corresponds exactly to a referenced commit in a Git repository available at the canonical location called *dgit-repos*, used by `dgit`, a bidirectional gateway between the Debian archive and Git. The commit is reachable from at least one reference whose name matches `refs/dgit/*`. See the manual page of `dgit` for further details.

5.7 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 Debian source 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 Debian source 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 Debian source control files will contain the field

```
Comment: I stand between the candle and the star.
```

5.8 Obsolete fields

The following fields have been obsoleted and may be found in packages conforming with previous versions of the Policy.

5.8.1 `DM-Upload-Allowed`

Indicates that Debian Maintainers may upload this package to the Debian archive. The only valid value is `yes`. This field was used to regulate uploads by Debian Maintainers, See the General Resolution Endorse the concept of Debian Maintainers (`http://www.debian.org/vote/2007/vote_003`) for more details.

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 control information files `preinst`, `postinst`, `prerm` and `postrm`. 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 must not be 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 exit with a zero status if everything went well.

Additionally, packages interacting with users using `debconf` in the `postinst` script should install a `config` script as a control information file. See 'Prompting in maintainer scripts' on page 12 for details.

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 unpacked, 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`, and `update-rc.d` can be found via the `PATH` environment variable. Those programs, and any other program that one would expect to be in 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]

[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.

6.3 Controlling terminal for maintainer scripts

Maintainer scripts are not guaranteed to run with a controlling terminal and may not be able to interact with the user. They must be able to fall back to noninteractive behavior if no controlling terminal is available. Maintainer scripts that prompt via a program conforming to the Debian Configuration Management Specification (see 'Prompting in maintainer scripts' on page 12) may assume that program will handle falling back to noninteractive behavior.

For high-priority prompts without a reasonable default answer, maintainer scripts may abort if there is no controlling terminal. However, this situation should be avoided if at all possible, since it prevents automated or unattended installs. In most cases, users will consider this to be a bug in the package.

6.4 Exit status

Each script must return a zero exit status for success, or a nonzero one for failure, since the package management system looks for the exit status of these scripts and determines what action to take next based on that datum.

6.5 Summary of ways maintainer scripts are called

What follows is a summary of all the ways in which maintainer scripts may be called along with what facilities those scripts may rely on being available at that time. Script names preceded by *new-* are the scripts from the new version of a package being installed, upgraded to, or downgraded to. Script names preceded by *old-* are the scripts from the old version of a package that is being upgraded from or downgraded from.

The `preinst` script may be called in the following ways:

***new-preinst* `install`**

new-preinst* `install` *old-version

new-preinst* `upgrade` *old-version The package will not yet be unpacked, so the `preinst` script cannot rely on any files included in its package. Only essential packages and pre-dependencies (`Pre-Depends`) may be assumed to be available. Pre-dependencies will have been configured at least once, but at the time the `preinst` is called they may only be in an "Unpacked" or "Half-Configured" state if a previous version of the pre-dependency was completely configured and has not been removed since then.

old-preinst* `abort-upgrade` *new-version Called during error handling of an upgrade that failed after unpacking the new package because the `postrm upgrade` action failed. The unpacked files may be partly from the new version or partly missing, so the script cannot rely on files included in the package. Package dependencies may not be available. Pre-dependencies will be at least "Unpacked" following the same rules as above, except they may be only "Half-Installed" if an upgrade of the pre-dependency failed.[2]

The `postinst` script may be called in the following ways:

postinst* `configure` *most-recently-configured-version The files contained in the package will be unpacked. All package dependencies will at least be "Unpacked". If there are no circular dependencies involved, all package dependencies will be configured. For behavior in the case of circular dependencies, see the discussion in 'Binary Dependencies - `Depends`, `Recommends`, `Suggests`, `Enhances`, `Pre-Depends`' on page 42.

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

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

***postinst* `abort-remove`**

deconfigured's-postinst* `abort-deconfigure in-favour` *failed-install-package version* `[removing` *conflicting-package versi
] The files contained in the package will be unpacked. All package dependencies will at least be "Half-Installed" and will have previously been configured and not removed. However, dependencies may not be configured or even fully unpacked in some error situations.[3] The `postinst` should still attempt any actions for which its dependencies are

[2]This can happen if the new version of the package no longer pre-depends on a package that had been partially upgraded.

[3]For example, suppose packages foo and bar are "Installed" with foo depending on bar. If an upgrade of bar were started and then aborted, and then an attempt to remove foo failed because its `prerm` script failed, foo's `postinst abort-remove` would be called with bar only "Half-Installed".

required, since they will normally be available, but consider the correct error handling approach if those actions fail. Aborting the `postinst` action if commands or facilities from the package dependencies are not available is often the best approach.

The `prerm` script may be called in the following ways:

prerm **`remove`**

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

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

deconfigured's-prerm **`deconfigure in-favour`** *package-being-installed version* [**`removing`** *conflicting-package version*
] The package whose `prerm` is being called will be at least "Half-Installed". All package dependencies will at least be "Half-Installed" and will have previously been configured and not removed. If there was no error, all dependencies will at least be "Unpacked", but these actions may be called in various error states where dependencies are only "Half-Installed" due to a partial upgrade.

new-prerm **`failed-upgrade`** *old-version* Called during error handling when `prerm upgrade` fails. The new package will not yet be unpacked, and all the same constraints as for `preinst upgrade` apply.

The `postrm` script may be called in the following ways:

postrm **`remove`**

postrm **`purge`**

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

disappearer's-postrm **`disappear`** *overwriter overwriter-version* The `postrm` script is called after the package's files have been removed or replaced. The package whose `postrm` is being called may have previously been deconfigured and only be "Unpacked", at which point subsequent package changes do not consider its dependencies. Therefore, all `postrm` actions may only rely on essential packages and must gracefully skip any actions that require the package's dependencies if those dependencies are unavailable.[4]

new-postrm **`failed-upgrade`** *old-version* Called when the old `postrm upgrade` action fails. The new package will be unpacked, but only essential packages and pre-dependencies can be relied on. Pre-dependencies will either be configured or will be "Unpacked" or "Half-Configured" but previously had been configured and was never removed.

new-postrm **`abort-install`**

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

new-postrm **`abort-upgrade`** *old-version* Called before unpacking the new package as part of the error handling of `preinst` failures. May assume the same state as `preinst` can assume.

6.6 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 1 If a version of the package is already "Installed", call

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

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

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

If this works, the upgrade continues. If this does not work, the error unwind:

[4]This is often done by checking whether the command or facility the `postrm` intends to call is available before calling it. For example:

```
if [ "$1" = purge ] && [ -e /usr/share/debconf/confmodule ]; then . /usr/share/debconf/confmodule db_purge fi
```

in `postrm` purges the `debconf` configuration for the package if `debconf` is installed.

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

If this works, then the old-version is "Installed", if not, the old version is in a "Half-Configured" state.

2 If a "conflicting" package is being removed at the same time, or if any package will be broken (due to `Breaks`):

 1 If `--auto-deconfigure` is specified, call, for each package to be deconfigured due to `Breaks`:

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

 Error unwind:

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

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

 2 If any packages depended on a conflicting package being removed 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.

 3 To prepare for removal of each 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 1 If the package is being upgraded, call:

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

 If this fails, we call:

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

  1 If that works, then

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

  is called. If this works, then the old version is in an "Installed" state, or else it is left in an "Unpacked" state.

  2 If it fails, then the old version is left in an "Half-Installed" state.

 2 Otherwise, if the package had some configuration files from a previous version installed (i.e., it is in the "Config-Files" state):

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

 Error unwind:

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

 If this fails, the package is left in a "Half-Installed" state, which requires a reinstall. If it works, the packages is left in a "Config-Files" state.

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

```
new-preinst install
```

 Error unwind:

```
new-postrm abort-install
```

 If the error-unwind fails, the package is in a "Half-Installed" phase, and requires a reinstall. If the error unwind works, the package is in the "Not-Installed" state.

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 contain files which are on the system in another package, unless `Replaces` is used (see 'Overwriting files and replacing packages - `Replaces`' on page 45).

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 unpacked which overwrites a file from another package, and is then removed again.[5]

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.

5. 1. If the package is being upgraded, call

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

   2. If this fails, `dpkg` will attempt:

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

   If this works, installation continues. If not, Error unwind:

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

   If this fails, the old version is left in a "Half-Installed" state. If it works, dpkg now calls:

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

   If this fails, the old version is left in a "Half-Installed" state. If it works, dpkg now calls:

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

   If this fails, the old version is in an "Unpacked" state.

   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

   1. `dpkg` calls:

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

   2. The package's maintainer scripts are removed.

   3. 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 unpacked have already been removed from the conflicting package's file list, and so do not get removed now).

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

6.7 Details of configuration

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

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

No attempt is made to unwind after errors during configuration. If the configuration fails, the package is in a "Half-Configured" state, and an error message is generated.

If there is no most recently configured version `dpkg` will pass a null argument. [6]

6.8 Details of removal and/or configuration purging

1
```
prerm remove
```

If prerm fails during replacement due to conflict

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

Or else we call:

```
postinst abort-remove
```

If this fails, the package is in a "Half-Configured" state, or else it remains "Installed".

2 The package's files are removed (except `conffile`s).

3
```
postrm remove
```

If it fails, there's no error unwind, and the package is in an "Half-Installed" state.

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 `conffile`s are automatically purged when removed, as there is no difference except for the `dpkg` status.

5 The `conffile`s and any backup files (`~`-files, `#*#` files, `%`-files, `.dpkg-{old,new,tmp}`, etc.) are removed.

6
```
postrm purge
```

If this fails, the package remains in a "Config-Files" state.

7 The package's file list is removed.

[6] Historical note: Truly ancient (pre-1997) versions of `dpkg` passed `<unknown>` (including the angle brackets) in this case. Even older ones did not pass a second argument at all, under any circumstance. Note that upgrades using such an old dpkg version are unlikely to work for other reasons, even if this old argument behavior is handled by your postinst script.

Chapter 7

Declaring relationships between packages

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 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, that part of the dependency can be satisfied by any one of the alternative packages.

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`' on page 28.

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 confusingly used to mean earlier/later or equal, rather than strictly earlier/later, and must not appear in new packages (though `dpkg` still supports them with a warning).

Whitespace may appear at any point in the version specification subject to the rules in 'Syntax of control files' on page 23, and must appear where it's necessary to disambiguate; it is not otherwise significant. All of the relationship fields can only be folded in source package control files. 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. When opening a continuation line in a relationship field, it is conventional to do so after a comma and before the space following that comma.

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
```

Relationships 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 non-empty list of Debian architecture names in the format described in 'Architecture specification strings' on page 79, 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 while others aren't.)

For build relationship fields (`Build-Depends`, `Build-Depends-Indep`, `Build-Conflicts` and `Build-Conflicts-Indep`), 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]
```

requires `kernel-headers-2.2.10` on all architectures other than hurd-i386 and requires `hurd-dev` and `gnumach-dev` only on hurd-i386.

For binary relationship fields and the `Built-Using` field, the architecture restriction syntax is only supported in the source package control file `debian/control`. When the corresponding binary package control file is generated, the relationship will either be omitted or included without the architecture restriction based on the architecture of the binary package. This means that architecture restrictions must not be used in binary relationship fields for architecture-independent packages (`Architecture: all`).

For example:

```
Depends: foo [i386], bar [amd64]
```

becomes `Depends: foo` when the package is built on the `i386` architecture, `Depends: bar` when the package is built on the `amd64` architecture, and omitted entirely in binary packages built on all other architectures.

If the architecture-restricted dependency is part of a set of alternatives using `|`, that alternative is ignored completely on architectures that do not match the restriction. For example:

```
Build-Depends: foo [!i386] | bar [!amd64]
```

is equivalent to `bar` on the i386 architecture, to `foo` on the amd64 architecture, and to `foo | bar` on all other architectures.

Relationships may also be restricted to a certain set of architectures using architecture wildcards in the format described in 'Architecture wildcards' on page 79. The syntax for declaring such restrictions is the same as declaring restrictions using a certain set of architectures without architecture wildcards. For example:

```
Build-Depends: foo [linux-any], bar [any-i386], baz [!linux-any]
```

is equivalent to `foo` on architectures using the Linux kernel and any cpu, `bar` on architectures using any kernel and an i386 cpu, and `baz` on any architecture using a kernel other than Linux.

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`

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.

This is done using the `Depends`, `Pre-Depends`, `Recommends`, `Suggests`, `Enhances`, `Breaks` and `Conflicts` control fields. `Breaks` is described in 'Packages which break other packages - `Breaks`' on the next page, and `Conflicts` is described in 'Conflicting binary packages - `Conflicts`' on page 44. The rest are described below.

These seven fields are used to declare a dependency relationship by one package on another. Except for `Enhances` and `Breaks`, they appear in the depending (binary) package's control file. (`Enhances` appears in the recommending package's control file, and `Breaks` appears in the version of depended-on package which causes the named package to break).

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 `apt-get`, `aptitude`, and `dselect`.)

Since `Depends` only places requirements on the order in which packages are configured, packages in an installation run are usually all unpacked first and all configured later. [1]

If there is a circular dependency among packages being installed or removed, installation or removal order honoring the dependency order is impossible, requiring the dependency loop be broken at some point and the dependency requirements violated for at least one package. Packages involved in circular dependencies may not be able to rely on their dependencies being configured before they themselves are configured, depending on which side of the break of the circular dependency loop they happen to be on. If one of the packages in the loop has no `postinst` script, then the cycle will be broken at that package; this ensures that all `postinst` scripts are run with their dependencies properly configured if this is possible. Otherwise the breaking point is arbitrary. Packages should therefore avoid circular dependencies where possible, particularly if they have `postinst` scripts.

The meaning of the five dependency fields is as follows:

[1] This approach makes dependency resolution easier. If two packages A and B are being upgraded, the installed package A depends on exactly the installed package B, and the new package A depends on exactly the new package B (a common situation when upgrading shared libraries and their corresponding development packages), satisfying the dependencies at every stage of the upgrade would be impossible. This relaxed restriction means that both new packages can be unpacked together and then configured in their dependency order.

`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 (unless there is a circular dependency as described above).

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` or `prerm` scripts require the depended-on package to be unpacked or configured in order to run. In the case of `postinst configure`, the depended-on packages will be unpacked and configured first. (If both packages are involved in a dependency loop, this might not work as expected; see the explanation a few paragraphs back.) In the case of `prerm` or other `postinst` actions, the package dependencies will normally be at least unpacked, but they may be only "Half-Installed" if a previous upgrade of the dependency failed.

Finally, the `Depends` field should be used if the depended-on package is needed by the `postrm` script to fully clean up after the package removal. There is no guarantee that package dependencies will be available when `postrm` is run, but the depended-on package is more likely to be available if the package declares a dependency (particularly in the case of `postrm remove`). The `postrm` script must gracefully skip actions that require a dependency if that dependency isn't available.

`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 in the "Unpacked" or the "Half-Configured" state, 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`. It will be considered satisfied only if the depended-on package has been correctly configured. However, unlike with `Depends`, `Pre-Depends` does not permit circular dependencies to be broken. If a circular dependency is encountered while attempting to honor `Pre-Depends`, the installation will be aborted.

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

`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.

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. See 'Dependencies' on page 11.

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 Packages which break other packages - `Breaks`

When one binary package declares that it breaks another, `dpkg` will refuse to allow the package which declares `Breaks` to be unpacked unless the broken package is deconfigured first, and it will refuse to allow the broken package to be reconfigured.

A package will not be regarded as causing breakage merely because its configuration files are still installed; it must be at least "Half-Installed".

A special exception is made for packages which declare that they break their own package name or a virtual package which they provide (see below): this does not count as a real breakage.

Normally a `Breaks` entry will have an "earlier than" version clause; such a `Breaks` is introduced in the version of an (implicit or explicit) dependency which violates an assumption or reveals a bug in earlier versions of the broken package, or which takes over a file from earlier versions of the package named in `Breaks`. This use of `Breaks` will inform higher-level package management tools that the broken package must be upgraded before the new one.

If the breaking package also overwrites some files from the older package, it should use `Replaces` to ensure this goes smoothly. See 'Overwriting files and replacing packages - `Replaces`' on the facing page for a full discussion of taking over files from other packages, including how to use `Breaks` in those cases.

Many of the cases where `Breaks` should be used were previously handled with `Conflicts` because `Breaks` did not yet exist. Many `Conflicts` fields should now be `Breaks`. See 'Conflicting binary packages - `Conflicts`' on the current page for more information about the differences.

7.4 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 unpacked on the system at the same time. This is a stronger restriction than `Breaks`, which prevents the broken package from being configured while the breaking package is in the "Unpacked" state but allows both packages to be unpacked at the same time.

If one package is to be unpacked, the other must be removed first. If the package being unpacked is marked as replacing (see 'Overwriting files and replacing packages - `Replaces`' on the facing page, but note that `Breaks` should normally be used in this case) 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.

Normally, `Breaks` should be used instead of `Conflicts` since `Conflicts` imposes a stronger restriction on the ordering of package installation or upgrade and can make it more difficult for the package manager to find a correct solution to an upgrade or installation problem. `Breaks` should be used

- when moving a file from one package to another (see 'Overwriting files and replacing packages - `Replaces`' on the next page),
- when splitting a package (a special case of the previous one), or
- when the breaking package exposes a bug in or interacts badly with particular versions of the broken package.

`Conflicts` should be used

- when two packages provide the same file and will continue to do so,
- in conjunction with `Provides` when only one package providing a given virtual facility may be unpacked at a time (see 'Virtual packages - `Provides`' on the facing page),
- in other cases where one must prevent simultaneous installation of two packages for reasons that are ongoing (not fixed in a later version of one of the packages) or that must prevent both packages from being unpacked at the same time, not just configured.

Be aware that adding `Conflicts` is normally not the best solution when two packages provide the same files. Depending on the reason for that conflict, using alternatives or renaming the files is often a better approach. See, for example, 'Binaries' on page 71.

Neither `Breaks` nor `Conflicts` should be used unless two packages cannot be installed at the same time or installing them both causes one of them to be broken or unusable. Having similar functionality or performing the same tasks as another package is not sufficient reason to declare `Breaks` or `Conflicts` with that package.

A `Conflicts` entry may have an "earlier than" version clause if the reason for the conflict is corrected in a later version of one of the packages. However, normally the presence of an "earlier than" version clause is a sign that `Breaks` should have been used instead. An "earlier than" version clause in `Conflicts` prevents `dpkg` from upgrading or installing the package which declares such a conflict until the upgrade or removal of the conflicted-with package has been completed, which is a strong restriction.

7.5 Virtual packages - `Provides`

As well as the names of actual ("concrete") packages, the package relationship fields `Depends`, `Recommends`, `Suggests`, `Enhances`, `Pre-Depends`, `Breaks`, `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 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. (See also 'Virtual packages' on page 11)

If there are both concrete and virtual packages of the same name, then the dependency may be satisfied (or the conflict caused) by either the concrete package with the name in question or any other concrete package which provides the virtual package with the name in question. This is so that, for example, supposing we have

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

and someone else releases an enhanced version of the `bar` package 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 relationship field has a version number attached, only real packages will be considered to see whether the relationship is satisfied (or the prohibition violated, for a conflict or breakage). In other words, if a version number is specified, this is a request to ignore all `Provides` for that package name and consider only real packages. The package manager will assume that a package providing that virtual package is not of the "right" version. 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 considered when considering a dependency on or conflict with the virtual package name.[2]

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

If the virtual package represents a facility that can only be provided by one real package at a time, such as the `mail-transport-agent` virtual package that requires installation of a binary that would conflict with all other providers of that virtual package (see 'Mail transport, delivery and user agents' on page 81), all packages providing that virtual package should also declare a conflict with it using `Conflicts`. This will ensure that at most one provider of that virtual package is unpacked or installed at a time.

7.6 Overwriting files and replacing packages - `Replaces`

Packages can declare in their control file that they should overwrite files in certain other packages, or completely replace other packages. The `Replaces` control field has these two distinct purposes.

7.6.1 Overwriting files in other packages

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 and will be taken over by the new package. Normally, `Breaks` should be used in conjunction with `Replaces`.[3]

For example, if a package `foo` is split into `foo` and `foo-data` starting at version 1.2-3, `foo-data` would have the fields

[2]It is possible that a future release of `dpkg` may add the ability 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.

[3]To see why `Breaks` is normally needed in addition to `Replaces`, consider the case of a file in the package `foo` being taken over by the package `foo-data`. `Replaces` will allow `foo-data` to be installed and take over that file. However, without `Breaks`, nothing requires `foo` to be upgraded to a newer version that knows it does not include that file and instead depends on `foo-data`. Nothing would prevent the new `foo-data` package from being installed and then removed, removing the file that it took over from `foo`. After that operation, the package manager would think the system was in a consistent state, but the `foo` package would be missing one of its files.

```
Replaces: foo (<< 1.2-3)
Breaks: foo (<< 1.2-3)
```

in its control file. The new version of the package `foo` would normally have the field

```
Depends: foo-data (>= 1.2-3)
```

(or possibly `Recommends` or even `Suggests` if the files moved into `foo-data` are not required for normal operation).

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 36. [4]

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

This usage of `Replaces` only takes effect when both packages are at least partially on the system at once. It is not relevant if the packages conflict unless the conflict has been overridden.

7.6.2 Replacing whole packages, forcing their removal

Second, `Replaces` allows the packaging system to resolve which package should be removed when there is a conflict (see 'Conflicting binary packages - `Conflicts`' on page 44). 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 unpacked at any one time. See 'Virtual packages - `Provides`' on the previous page for more information about this example.

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

Source packages that require certain binary packages to be installed or absent at the time of building the package can declare relationships to those binary packages.

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

Build-dependencies on "build-essential" binary packages can be omitted. Please see 'Package relationships' on page 15 for more information.

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:[5]

`clean`, `build-arch`, and `binary-arch` Only the `Build-Depends` and `Build-Conflicts` fields must be satisfied when these targets are invoked.

`build`, `build-indep`, `binary`, and `binary-indep` The `Build-Depends`, `Build-Conflicts`, `Build-Depends-Indep`, and `Build-Conflicts-Indep` fields must be satisfied when these targets are invoked.

[4]Replaces is a one way relationship. You have to install the replacing package after the replaced package.

[5]There is no Build-Depends-Arch; this role is essentially met with Build-Depends. Anyone building the `build-indep` and `binary-indep` targets is assumed to be building the whole package, and therefore installation of all build dependencies is required. The autobuilders use `dpkg-buildpackage -B`, which calls `build`, not `build-arch` since it does not yet know how to check for its existence, and `binary-arch`. The purpose of the original split between `Build-Depends` and `Build-Depends-Indep` was so that the autobuilders wouldn't need to install extra packages needed only for the binary-indep targets. But without a build-arch/build-indep split, this didn't work, since most of the work is done in the build target, not in the binary target.

7.8 Additional source packages used to build the binary - `Built-Using`

Some binary packages incorporate parts of other packages when built but do not have to depend on those packages. Examples include linking with static libraries or incorporating source code from another package during the build. In this case, the source packages of those other packages are a required part of the complete source (the binary package is not reproducible without them).

A `Built-Using` field must list the corresponding source package for any such binary package incorporated during the build [6], including an "exactly equal" ("=") version relation on the version that was used to build that binary package[7].

A package using the source code from the gcc-4.6-source binary package built from the gcc-4.6 source package would have this field in its control file:

```
Built-Using: gcc-4.6 (= 4.6.0-11)
```

A package including binaries from grub2 and loadlin would have this field in its control file:

```
Built-Using: grub2 (= 1.99-9), loadlin (= 1.6e-1)
```

[6] `Build-Depends` in the source package is not adequate since it (rightfully) does not document the exact version used in the build.
[7] The archive software might reject packages that refer to non-existent sources.

Chapter 8

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`).

This section deals only with public shared libraries: shared libraries that are placed in directories searched by the dynamic linker by default or which are intended to be linked against normally and possibly used by other, independent packages. Shared libraries that are internal to a particular package or that are only loaded as dynamic modules are not covered by this section and are not subject to its requirements.

A shared library is identified by the `SONAME` attribute stored in its dynamic section. When a binary is linked against a shared library, the `SONAME` of the shared library is recorded in the binary's `NEEDED` section so that the dynamic linker knows that library must be loaded at runtime. The shared library file's full name (which usually contains additional version information not needed in the `SONAME`) is therefore normally not referenced directly. Instead, the shared library is loaded by its `SONAME`, which exists on the file system as a symlink pointing to the full name of the shared library. This symlink must be provided by the package. 'Run-time shared libraries' on the current page describes how to do this. [1]

When linking a binary or another shared library against a shared library, the `SONAME` for that shared library is not yet known. Instead, the shared library is found by looking for a file matching the library name with `.so` appended. This file exists on the file system as a symlink pointing to the shared library.

Shared libraries are normally split into several binary packages. The `SONAME` symlink is installed by the runtime shared library package, and the bare `.so` symlink is installed in the development package since it's only used when linking binaries or shared libraries. However, there are some exceptions for unusual shared libraries or for shared libraries that are also loaded as dynamic modules by other programs.

This section is primarily concerned with how the separation of shared libraries into multiple packages should be done and how dependencies on and between shared library binary packages are managed in Debian. 'Libraries' on page 71 should be read in conjunction with this section and contains additional rules for the files contained in the shared library packages.

8.1 Run-time shared libraries

The run-time shared library must be placed in a package whose name changes whenever the `SONAME` of the shared library changes. This allows several versions of the shared library to be installed at the same time, allowing installation of the new version of the shared library without immediately breaking binaries that depend on the old version. Normally, the run-time shared library and its `SONAME` symlink should be placed in a package named *`librarynamesoversion`*, where *soversion* is the version number in the `SONAME` of the shared library. Alternatively, if it would be confusing to directly append *soversion* to *libraryname* (if, for example, *libraryname* itself ends in a number), you should use *`libraryname-soversion`* instead.

To determine the *soversion*, look at the `SONAME` of the library, stored in the ELF `SONAME` attribute. It is usually of the form *`name`*`.so.`*`major-version`* (for example, `libz.so.1`). The version part is the part which comes after `.so.`, so in that example it is `1`. The soname may instead be of the form *`name-major-version`*`.so`, such as `libdb-5.1.so`, in which case the name would be `libdb` and the version would be `5.1`.

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 all of their `SONAME`s will always change together. Be aware that this is not normally the case,

[1]This is a convention of shared library versioning, but not a requirement. Some libraries use the `SONAME` as the full library file name instead and therefore do not need a symlink. Most, however, encode additional information about backwards-compatible revisions as a minor version number in the file name. The `SONAME` itself only changes when binaries linked with the earlier version of the shared library may no longer work, but the filename may change with each release of the library. See 'Run-time shared libraries' on this page for more information.

and if the `SONAME`s do not change together, upgrading such a merged shared library package will be unnecessarily difficult because of file conflicts with the old version of the package. When in doubt, always split shared library packages so that each binary package installs a single shared library.

Every time the shared library ABI changes in a way that may break binaries linked against older versions of the shared library, the `SONAME` of the library and the corresponding name for the binary package containing the runtime shared library should change. Normally, this means the `SONAME` should change any time an interface is removed from the shared library or the signature of an interface (the number of parameters or the types of parameters that it takes, for example) is changed. This practice is vital to allowing clean upgrades from older versions of the package and clean transitions between the old ABI and new ABI without having to upgrade every affected package simultaneously.

The `SONAME` and binary package name need not, and indeed normally should not, change if new interfaces are added but none are removed or changed, since this will not break binaries linked against the old shared library. Correct versioning of dependencies on the newer shared library by binaries that use the new interfaces is handled via the `symbols` or `shlibs` system.

The package should install the shared libraries under their normal names. For example, the `libgdbm3` package should install `libgdbm.so.3.0.0` as `/usr/lib/libgdbm.so.3.0.0`. 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.

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.

The run-time library package should include the symbolic link for the `SONAME` that `ldconfig` would create for the shared libraries. For example, the `libgdbm3` package should include a symbolic link from `/usr/lib/libgdbm.so.3` to `libgdbm.so.3.0.0`. 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.[2]

8.1.1 `ldconfig`

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`[3] must use `ldconfig` to update the shared library system.

The package maintainer scripts must only call `ldconfig` under these circumstances:

- When the `postinst` script is run with a first argument of `configure`, the script must call `ldconfig`, and may optionally invoke `ldconfig` at other times.
- When the `postrm` script is run with a first argument of `remove`, the script should call `ldconfig`.

[4]

8.2 Shared library support files

If your package contains files whose names do not change with each change in the library shared object version, you must not put them in the shared library package. Otherwise, several versions of the shared library cannot be installed at the same time without filename clashes, making upgrades and transitions unnecessarily difficult.

It is recommended that supporting files and run-time support programs that do not need to be invoked manually by users, but are nevertheless required for the package to function, be placed (if they are binary) in a subdirectory of `/usr/lib`,

[2]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. Since version 1.7.0, `dpkg` reorders the files itself as necessary when building a package. Thus it is no longer important to concern oneself with the order of file creation.

[3]These are currently `/usr/local/lib` plus directories under `/lib` and `/usr/lib` matching the multiarch triplet for the system architecture.

[4]During install or upgrade, the preinst is called before the new files are unpacked, so calling "ldconfig" is pointless. The preinst of an existing package can also be called if an upgrade fails. However, this happens during the critical time when a shared libs may exist on-disk under a temporary name. Thus, it is dangerous and forbidden by current policy to call "ldconfig" at this time. When a package is installed or upgraded, "postinst configure" runs after the new files are safely on-disk. Since it is perfectly safe to invoke ldconfig unconditionally in a postinst, it is OK for a package to simply put ldconfig in its postinst without checking the argument. The postinst can also be called to recover from a failed upgrade. This happens before any new files are unpacked, so there is no reason to call "ldconfig" at this point. For a package that is being removed, prerm is called with all the files intact, so calling ldconfig is useless. The other calls to "prerm" happen in the case of upgrade at a time when all the files of the old package are on-disk, so again calling "ldconfig" is pointless. postrm, on the other hand, is called with the "remove" argument just after the files are removed, so this is the proper time to call "ldconfig" to notify the system of the fact that the shared libraries from the package are removed. The postrm can be called at several other times. At the time of "postrm purge", "postrm abort-install", or "postrm abort-upgrade", calling "ldconfig" is useless because the shared lib files are not on-disk. However, when "postrm" is invoked with arguments "upgrade", "failed-upgrade", or "disappear", a shared lib may exist on-disk under a temporary filename.

preferably under `/usr/lib/`*package-name*. If the program or file is architecture independent, the recommendation is for it to be placed in a subdirectory of `/usr/share` instead, preferably under `/usr/share/`*package-name*. Following the *package-name* naming convention ensures that the file names change when the shared object version changes.

Run-time support programs that use the shared library but are not required for the library to function or files used by the shared library that can be used by any version of the shared library package should instead be put in a separate package. This package might typically be named `libraryname-tools`; note the absence of the *soversion* in the package name.

Files and support programs only useful when compiling software against the library should be included in the development package for the library.[5]

8.3 Static libraries

The static library (`libraryname.a`) is usually provided in addition to the shared version. It is placed into the development package (see below).

In some cases, it is acceptable for a library to be available in static form only; these cases include:

- libraries for languages whose shared library support is immature or unstable
- libraries whose interfaces are in flux or under development (commonly the case when the library's major version number is zero, or where the ABI breaks across patchlevels)
- libraries which are explicitly intended to be available only in static form by their upstream author(s)

8.4 Development files

If there are development files associated with a shared library, the source package needs to generate a binary development package named `librarynamesoversion-dev`, or if you prefer only to support one development version at a time, `libraryname-dev`. Installing the development package must result in installation of all the development files necessary for compiling programs against that shared library.[6]

In case several development versions of a library exist, you may need to use `dpkg`'s Conflicts mechanism (see 'Conflicting binary packages - `Conflicts`' on page 44) 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 unpacked).

The development package should contain a symlink for the associated shared library without a version number. For example, the `libgdbm-dev` package should include a symlink from `/usr/lib/libgdbm.so` to `libgdbm.so.3.0.0`. This symlink is needed by the linker (`ld`) when compiling packages, as it will only look for `libgdbm.so` when compiling dynamically.

If the package provides Ada Library Information (`*.ali`) files for use with GNAT, these files must be installed read-only (mode 0444) so that GNAT will not attempt to recompile them. This overrides the normal file mode requirements given in 'Permissions and owners' on page 77.

8.5 Dependencies between the packages of the same library

Typically the development version should have an exact version dependency on the runtime library, to make sure that compilation and linking happens correctly. The `${binary:Version}` substitution variable can be useful for this purpose. [7]

[5] For example, a `package-name-config` script or `pkg-config` configuration files.

[6] This wording allows the development files to be split into several packages, such as a separate architecture-independent `libraryname-headers`, provided that the development package depends on all the required additional packages.

[7] Previously, `${Source-Version}` was used, but its name was confusing and it has been deprecated since dpkg 1.13.19.

8.6 Dependencies between the library and other packages

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. These dependencies must be added to the binary package when it is built, since they may change based on which version of a shared library the binary or library was linked with even if there are no changes to the source of the binary (for example, symbol versions change, macros become functions or vice versa, or the binary package may determine at compile-time whether new library interfaces are available and can be called). To allow these dependencies to be constructed, shared libraries must provide either a `symbols` file or a `shlibs` file. These provide information on the package dependencies required to ensure the presence of interfaces provided by this library. Any package with binaries or libraries linking to a shared library must use these files to determine the required dependencies when it is built. Other packages which use a shared library (for example using `dlopen()`) should compute appropriate dependencies using these files at build time as well.

The two mechanisms differ in the degree of detail that they provide. A `symbols` file documents, for each symbol exported by a library, the minimal version of the package any binary using this symbol will need. This is typically the version of the package in which the symbol was introduced. This information permits detailed analysis of the symbols used by a particular package and construction of an accurate dependency, but it requires the package maintainer to track more information about the shared library.

A `shlibs` file, in contrast, only documents the last time the library ABI changed in any way. It only provides information about the library as a whole, not individual symbols. When a package is built using a shared library with only a `shlibs` file, the generated dependency will require a version of the shared library equal to or newer than the version of the last ABI change. This generates unnecessarily restrictive dependencies compared to `symbols` files if none of the symbols used by the package have changed. This, in turn, may make upgrades needlessly complex and unnecessarily restrict use of the package on systems with older versions of the shared libraries.

`shlibs` files also only support a limited range of library SONAMEs, making it difficult to use `shlibs` files in some unusual corner cases.[8]

`symbols` files are therefore recommended for most shared library packages since they provide more accurate dependencies. For most C libraries, the additional detail required by `symbols` files is not too difficult to maintain. However, maintaining exhaustive symbols information for a C++ library can be quite onerous, so `shlibs` files may be more appropriate for most C++ libraries. Libraries with a corresponding udeb must also provide a `shlibs` file, since the udeb infrastructure does not use `symbols` files.

8.6.1 Generating dependencies on shared libraries

When a package that contains any shared libraries or compiled binaries is built, it must run `dpkg-shlibdeps` on each shared library and compiled binary to determine the libraries used and hence the dependencies needed by the package.[9] To do this, put a call to `dpkg-shlibdeps` into your `debian/rules` file in the source package. List all of the compiled binaries, libraries, or loadable modules in your package.[10] `dpkg-shlibdeps` will use the `symbols` or `shlibs` files installed by the shared libraries to generate dependency information. The package must then provide a substitution variable into which the discovered dependency information can be placed.

If you are creating a udeb for use in the Debian Installer, you will need to specify that `dpkg-shlibdeps` should use the dependency line of type `udeb` by adding the `-tudeb` option[11]. If there is no dependency line of type `udeb` in the `shlibs` file, `dpkg-shlibdeps` will fall back to the regular dependency line.

`dpkg-shlibdeps` puts the dependency information into the `debian/substvars` file by default, which is then used by `dpkg-gencontrol`. You will need to place a `${shlibs:Depends}` variable in the `Depends` field in the control file of every binary package built by this source package that contains compiled binaries, libraries, or loadable modules. If you have multiple binary packages, you will need to call `dpkg-shlibdeps` on each one which contains compiled libraries or binaries. For example, you could use the `-T` option to the `dpkg` utilities to specify a different `substvars` file for each binary package.[12]

For more details on `dpkg-shlibdeps`, see `dpkg-shlibdeps(1)`.

[8] A `shlibs` file represents an SONAME as a library name and version number, such as `libfoo VERSION`, instead of recording the actual SONAME. If the SONAME doesn't match one of the two expected formats (`libfoo-VERSION.so` or `libfoo.so.VERSION`), it cannot be represented.

[9] `dpkg-shlibdeps` will use a program like `objdump` or `readelf` to find the libraries and the symbols in those libraries directly needed by the binaries or shared libraries in the package.

[10] The easiest way to call `dpkg-shlibdeps` correctly is to use a package helper framework such as `debhelper`. If you are using `debhelper`, the `dh_shlibdeps` program will do this work for you. It will also correctly handle multi-binary packages.

[11] `dh_shlibdeps` from the `debhelper` suite will automatically add this option if it knows it is processing a udeb.

[12] Again, `dh_shlibdeps` and `dh_gencontrol` will handle everything except the addition of the variable to the control file for you if you're using `debhelper`, including generating separate `substvars` files for each binary package and calling `dpkg-gencontrol` with the appropriate flags.

We say that a binary foo *directly* uses a library libbar if it is explicitly linked with that library (that is, the library is listed in the ELF NEEDED attribute, caused by adding -lbar to the link line when the binary is created). 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 depend on the libraries it directly uses, but not the libraries it only uses indirectly. The dependencies for the libraries used directly will automatically pull in the indirectly-used libraries. dpkg-shlibdeps will handle this logic automatically, but package maintainers need to be aware of this distinction between directly and indirectly using a library if they have to override its results for some reason. [13]

8.6.2 Shared library ABI changes

Maintaining a shared library package using either symbols or shlibs files requires being aware of the exposed ABI of the shared library and any changes to it. Both symbols and shlibs files record every change to the ABI of the shared library; symbols files do so per public symbol, whereas shlibs files record only the last change for the entire library.

There are two types of ABI changes: ones that are backward-compatible and ones that are not. An ABI change is backward-compatible if any reasonable program or library that was linked with the previous version of the shared library will still work correctly with the new version of the shared library.[14] Adding new symbols to the shared library is a backward-compatible change. Removing symbols from the shared library is not. Changing the behavior of a symbol may or may not be backward-compatible depending on the change; for example, changing a function to accept a new enum constant not previously used by the library is generally backward-compatible, but changing the members of a struct that is passed into library functions is generally not unless the library takes special precautions to accept old versions of the data structure.

ABI changes that are not backward-compatible normally require changing the SONAME of the library and therefore the shared library package name, which forces rebuilding all packages using that shared library to update their dependencies and allow them to use the new version of the shared library. For more information, see 'Run-time shared libraries' on page 49. The remainder of this section will deal with backward-compatible changes.

Backward-compatible changes require either updating or recording the *minimal-version* for that symbol in symbols files or updating the version in the *dependencies* in shlibs files. For more information on how to do this in the two formats, see 'The symbols File Format' on the next page and 'The shlibs File Format' on page 56. Below are general rules that apply to both files.

The easy case is when a public symbol is added. Simply add the version at which the symbol was introduced (for symbols files) or update the dependency version (for shlibs) files. But special care should be taken to update dependency versions when the behavior of a public symbol changes. This is easy to neglect, since there is no automated method of determining such changes, but failing to update versions in this case may result in binary packages with too-weak dependencies that will fail at runtime, possibly in ways that can cause security vulnerabilities. If the package maintainer believes that a symbol behavior change may have occurred but isn't sure, it's safer to update the version rather than leave it unmodified. This may result in unnecessarily strict dependencies, but it ensures that packages whose dependencies are satisfied will work properly.

A common example of when a change to the dependency version is required is a function that takes an enum or struct argument that controls what the function does. For example:

```
enum library_op { OP_FOO, OP_BAR };
int library_do_operation(enum library_op);
```

If a new operation, OP_BAZ, is added, the *minimal-version* of library_do_operation (for symbols files) or the version in the dependency for the shared library (for shlibs files) must be increased to the version at which OP_BAZ was introduced. Otherwise, a binary built against the new version of the library (having detected at compile-time that the library supports OP_BAZ) may be installed with a shared library that doesn't support OP_BAZ and will fail at runtime when it tries to pass OP_BAZ into this function.

Dependency versions in either symbols or shlibs files normally should not contain the Debian revision of the package, since the library behavior is normally fixed for a particular upstream version and any Debian packaging of that upstream version will have the same behavior. In the rare case that the library behavior was changed in a particular Debian revision, appending ~ to the end of the version that includes the Debian revision is recommended, since this allows backports of the shared library package using the normal backport versioning convention to satisfy the dependency.

[13] A good example of where this helps is the following. We could update libimlib with a new version that supports a new revision of a graphics format called dgf (but retaining the same major version number) and depends on a new library package libdgf4 instead of the older libdgf3. If we used ldd to add dependencies for every library directly or indirectly linked with a binary, every package that uses libimlib would need to be recompiled so it would also depend on libdgf4 in order to retire the older libdgf3 package. Since dependencies are only added based on ELF NEEDED attribute, packages using libimlib can rely on libimlib itself having the dependency on an appropriate version of libdgf and do not need rebuilding.

[14] An example of an "unreasonable" program is one that uses library interfaces that are documented as internal and unsupported. If the only programs or libraries affected by a change are "unreasonable" ones, other techniques, such as declaring Breaks relationships with affected packages or treating their usage of the library as bugs in those packages, may be appropriate instead of changing the SONAME. However, the default approach is to change the SONAME for any change to the ABI that could break a program.

8.6.3 The `symbols` system

In the following sections, we will first describe where the various `symbols` files are to be found, then the `symbols` file format, and finally how to create `symbols` files if your package contains a shared library.

The `symbols` files present on the system

`symbols` files for a shared library are normally provided by the shared library package as a control file, but there are several override paths that are checked first in case that information is wrong or missing. The following list gives them in the order in which they are read by `dpkg-shlibdeps` The first one that contains the required information is used.

- `debian/*/DEBIAN/symbols`

  During the package build, if the package itself contains shared libraries with `symbols` files, they will be generated in these staging directories by `dpkg-gensymbols` (see 'Providing a `symbols` file' on the next page). `symbols` files found in the build tree take precedence over `symbols` files from other binary packages.

  These files must exist before `dpkg-shlibdeps` is run or the dependencies of binaries and libraries from a source package on other libraries from that same source package will not be correct. In practice, this means that `dpkg-gensymbols` must be run before `dpkg-shlibdeps` during the package build.[15]

- `/etc/dpkg/symbols/`*`package`*`.symbols.`*`arch`* and `/etc/dpkg/symbols/`*`package`*`.symbols`

  Per-system overrides of shared library dependencies. These files normally do not exist. They are maintained by the local system administrator and must not be created by any Debian package.

- `symbols` control files for packages installed on the system

  The `symbols` control files for all the packages currently installed on the system are searched last. This will be the most common source of shared library dependency information. These are normally found in `/var/lib/dpkg/info/*.symbols`, but packages should not rely on this and instead should use `dpkg-query --control-path` *`package`* `symbols` if for some reason these files need to be examined.

Be aware that if a `debian/shlibs.local` exists in the source package, it will override any `symbols` files. This is the only case where a `shlibs` is used despite `symbols` files being present. See 'The `shlibs` files present on the system' on page 56 and 'The `shlibs` system' on page 56 for more information.

The `symbols` File Format

The following documents the format of the `symbols` control file as included in binary packages. These files are built from template `symbols` files in the source package by `dpkg-gensymbols`. The template files support a richer syntax that allows `dpkg-gensymbols` to do some of the tedious work involved in maintaining `symbols` files, such as handling C++ symbols or optional symbols that may not exist on particular architectures. When writing `symbols` files for a shared library package, refer to `dpkg-gensymbols(1)` for the richer syntax.

A `symbols` may contain one or more entries, one for each shared library contained in the package corresponding to that `symbols`. Each entry has the following format:

```
library-soname main-dependency-template
[| alternative-dependency-template]
[...]
[* field-name: field-value]
[...]
symbol minimal-version[ id-of-dependency-template ]
```

To explain this format, we'll use the the `zlib1g` package as an example, which (at the time of writing) installs the shared library `/usr/lib/libz.so.1.2.3.4`. Mandatory lines will be described first, followed by optional lines.

library-soname must contain exactly the value of the ELF `SONAME` attribute of the shared library. In our example, this is `libz.so.1`.[16]

[15] An example may clarify. Suppose the source package `foo` generates two binary packages, `libfoo2` and `foo-runtime`. When building the binary packages, the contents of the packages are staged 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 contain a `symbols` file, which will be installed in `debian/libfoo2/DEBIAN/symbols`, eventually to be included as a control file in that package. When `dpkg-shlibdeps` is run on the executable `debian/foo-runtime/usr/bin/foo-prog`, it will examine the `debian/libfoo2/DEBIAN/symbols` file to determine whether `foo-prog`'s library dependencies are satisfied by any of the libraries provided by `libfoo2`. Since those binaries were linked against the just-built shared library as part of the build process, the `symbols` file for the newly-built `libfoo2` must take precedence over a `symbols` file for any other `libfoo2` package already installed on the system.

[16] This can be determined by using the command

main-dependency-template has the same syntax as a dependency field in a binary package control file, except that the string `#MINVER#` is replaced by a version restriction like `(>= version)` or by nothing if an unversioned dependency is deemed sufficient. The version restriction will be based on which symbols from the shared library are referenced and the version at which they were introduced (see below). In nearly all cases, *main-dependency-template* will be `package #MINVER#`, where *package* is the name of the binary package containing the shared library. This adds a simple, possibly-versioned dependency on the shared library package. In some rare cases, such as when multiple packages provide the same shared library ABI, the dependency template may need to be more complex.

In our example, the first line of the `zlib1g` `symbols` file would be:

```
libz.so.1 zlib1g #MINVER#
```

Each public symbol exported by the shared library must have a corresponding symbol line, indented by one space. *symbol* is the exported symbol (which, for C++, means the mangled symbol) followed by `@` and the symbol version, or the string `Base` if there is no symbol version. *minimal-version* is the most recent version of the shared library that changed the behavior of that symbol, whether by adding it, changing its function signature (the parameters, their types, or the return type), or changing its behavior in a way that is visible to a caller. *id-of-dependency-template* is an optional field that references an *alternative-dependency-template*; see below for a full description.

For example, `libz.so.1` contains the symbols `compress` and `compressBound`. `compress` has no symbol version and last changed its behavior in upstream version `1:1.1.4`. `compressBound` has the symbol version `ZLIB_1.2.0`, was introduced in upstream version `1:1.2.0`, and has not changed its behavior. Its `symbols` file therefore contains the lines:

```
compress@Base 1:1.1.4
compressBound@ZLIB_1.2.0 1:1.2.0
```

Packages using only `compress` would then get a dependency on `zlib1g (>= 1:1.1.4)`, but packages using `compressBound` would get a dependency on `zlib1g (>= 1:1.2.0)`.

One or more *alternative-dependency-template* lines may be provided. These are used in cases where some symbols in the shared library should use one dependency template while others should use a different template. The alternative dependency templates are used only if a symbol line contains the *id-of-dependency-template* field. The first alternative dependency template is numbered 1, the second 2, and so forth.[17]

Finally, the entry for the library may contain one or more metadata fields. Currently, the only supported *field-name* is `Build-Depends-Package`, whose value lists the library development package on which packages using this shared library declare a build dependency. If this field is present, `dpkg-shlibdeps` uses it to ensure that the resulting binary package dependency on the shared library is at least as strict as the source package dependency on the shared library development package.[18] For our example, the `zlib1g` `symbols` file would contain:

```
* Build-Depends-Package: zlib1g-dev
```

Also see `deb-symbols(5)`.

Providing a `symbols` file

If your package provides a shared library, you should arrange to include a `symbols` control file following the format described above in that package. You must include either a `symbols` control file or a `shlibs` control file.

Normally, this is done by creating a `symbols` in the source package named `debian/package.symbols` or `debian/symbols`, possibly with `.arch` appended if the symbols information varies by architecture. This file may use the extended syntax documented in `dpkg-gensymbols(1)`. Then, call `dpkg-gensymbols` as part of the package build process. It will create `symbols` files in the package staging area based on the binaries and libraries in the package staging area and the `symbols` files in the source package.[19]

```
readelf -d /usr/lib/libz.so.1.2.3.4 | grep SONAME
```

[17] An example of where this may be needed is with a library that implements the libGL interface. All GL implementations provide the same set of base interfaces, and then may provide some additional interfaces only used by programs that require that specific GL implementation. So, for example, libgl1-mesa-glx may use the following `symbols` file:

```
libGL.so.1 libgl1 | libgl1-mesa-glx #MINVER# publicGlSymbol@Base 6.3-1 [...] implementationSpecificSymbol@Base 6.5.2-7 1 [...]
```

Binaries or shared libraries using only `publicGlSymbol` would depend only on `libgl1` (which may be provided by multiple packages), but ones using `implementationSpecificSymbol` would get a dependency on `libgl1-mesa-glx (>= 6.5.2-7)`

[18] This field should normally not be necessary, since if the behavior of any symbol has changed, the corresponding symbol *minimal-version* should have been increased. But including it makes the `symbols` system more robust by tightening the dependency in cases where the package using the shared library specifically requires at least a particular version of the shared library development package for some reason.

[19] If you are using `debhelper`, `dh_makeshlibs` will take care of calling either `dpkg-gensymbols` or generating a `shlibs` file as appropriate.

Packages that provide `symbols` files must keep them up-to-date to ensure correct dependencies in packages that use the shared libraries. This means updating the `symbols` file whenever a new public symbol is added, changing the *minimal-version* field whenever a symbol changes behavior or signature in a backward-compatible way (see 'Shared library ABI changes' on page 53), and changing the *library-soname* and *main-dependency-template*, and probably all of the *minimal-version* fields, when the library changes `SONAME`. Removing a public symbol from the `symbols` file because it's no longer provided by the library normally requires changing the `SONAME` of the library. See 'Run-time shared libraries' on page 49 for more information on `SONAME`s.

8.6.4 The `shlibs` system

The `shlibs` system is a simpler alternative to the `symbols` system for declaring dependencies for shared libraries. It may be more appropriate for C++ libraries and other cases where tracking individual symbols is too difficult. It predated the `symbols` system and is therefore frequently seen in older packages. It is also required for udebs, which do not support `symbols`.

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.

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. This file should normally not be used, but may be needed temporarily in unusual situations to work around bugs in other packages, or in unusual cases where the normally declared dependency information in the installed `shlibs` file for a library cannot be used. This file overrides information obtained from any other source.

- `/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"

  These files are generated as part of the package build process and staged for inclusion as control files in the binary packages being built. They provide details of any shared libraries included in the same package.

- `shlibs` control files for packages installed on the system

  The `shlibs` control files for all the packages currently installed on the system. These are normally found in `/var/lib/dpkg/info/*.shlibs`, but packages should not rely on this and instead should use `dpkg-query --control-path` *`package`* `shlibs` if for some reason these files need to be examined.

- `/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.

If a `symbols` file for a shared library package is available, `dpkg-shlibdeps` will always use it in preference to a `shlibs`, with the exception of `debian/shlibs.local`. The latter overrides any other `shlibs` or `symbols` files.

The `shlibs` File Format

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

```
[type: ]library-name soname-version 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.1.2.3.4`.

type is an optional element that indicates the type of package for which the line is valid. The only type currently in use is `udeb`. The colon and space after the type are required.

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 is the version part of the ELF `SONAME` attribute of the library, determined the same way that the *soversion* component of the recommended shared library package name is determined. See 'Run-time shared libraries' on page 49 for the details.

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 41 for details on the syntax, and 'Shared library ABI changes' on page 53 for details on how to maintain the dependency version constraint.

In our example, if the last change to the `zlib1g` package that could change behavior for a client of that library was in version `1:1.2.3.3.dfsg-1`, then the `shlibs` entry for this library could say:

```
libz 1 zlib1g (>= 1:1.2.3.3.dfsg)
```

This version restriction must be new enough that any binary built against the current version of the library will work with any version of the shared library that satisfies that dependency.

As zlib1g also provides a udeb containing the shared library, there would also be a second line:

```
udeb: libz 1 zlib1g-udeb (>= 1:1.2.3.3.dfsg)
```

Providing a `shlibs` file

To provide a `shlibs` file for a shared library binary package, create a `shlibs` file following the format described above and place it in the `DEBIAN` directory for that package during the build. It will then be included as a control file for that package[20].

Since `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.

[20]This is what `dh_makeshlibs` in the `debhelper` suite does. If your package also has a udeb that provides a shared library, `dh_makeshlibs` can automatically generate the `udeb:` lines if you specify the name of the udeb with the `--add-udeb` option.

Chapter 9

The Operating System

9.1 File system hierarchy

9.1.1 File System Structure

The location of all files and directories must comply with the Filesystem Hierarchy Standard (FHS), version 2.3, with the exceptions noted below, and except where doing so would violate other terms of Debian Policy. The following exceptions to the FHS apply:

1 The optional rules related to user specific configuration files for applications are stored in the user's home directory are relaxed. It is recommended that such files start with the '`.`' character (a "dot file"), and if an application needs to create more than one dot file then the preferred placement is in a subdirectory with a name starting with a '.' character, (a "dot directory"). In this case it is recommended the configuration files not start with the '.' character.

2 The requirement for amd64 to use `/lib64` for 64 bit binaries is removed.

3 The requirement for object files, internal binaries, and libraries, including `libc.so.*`, to be located directly under `/lib{,32}` and `/usr/lib{,32}` is amended, permitting files to instead be installed to `/lib/`*`triplet`* and `/usr/lib/`*`triplet`*, where *`triplet`* is the value returned by `dpkg-architecture -qDEB_HOST_MULTIARCH` for the architecture of the package. Packages may *not* install files to any *triplet* path other than the one matching the architecture of that package; for instance, an `Architecture: amd64` package containing 32-bit x86 libraries may not install these libraries to `/usr/lib/i386-linux-gnu`. [1]

Applications may also use a single subdirectory under `/usr/lib/`*`triplet`*.

The execution time linker/loader, ld*, must still be made available in the existing location under /lib or /lib64 since this is part of the ELF ABI for the architecture.

4 The requirement that `/usr/local/share/man` be "synonymous" with `/usr/local/man` is relaxed to a recommendation

5 The requirement that windowmanagers with a single configuration file call it `system.*wmrc` is removed, as is the restriction that the window manager subdirectory be named identically to the window manager name itself.

6 The requirement that boot manager configuration files live in `/etc`, or at least are symlinked there, is relaxed to a recommendation.

7 The additional directory `/run` in the root file system is allowed. `/run` replaces `/var/run`, and the subdirectory `/run/lock` replaces `/var/lock`, with the `/var` directories replaced by symlinks for backwards compatibility. `/run` and `/run/lock` must follow all of the requirements in the FHS for `/var/run` and `/var/lock`, respectively, such as file naming conventions, file format requirements, or the requirement that files be cleared during the boot process. Files and directories residing in `/run` should be stored on a temporary file system.

Packages must not assume the `/run` directory exists or is usable without a dependency on `initscripts (>= 2.88dsf-13.3)` until the stable release of Debian supports `/run`.

8 The `/sys` directory in the root filesystem is additionally allowed. [2]

[1]This is necessary in order to reserve the directories for use in cross-installation of library packages from other architectures, as part of the planned deployment of `multiarch`.

[2]This directory is used as mount point to mount virtual filesystems to get access to kernel information.

9 On GNU/Hurd systems, the following additional directories are allowed in the root filesystem: /hurd and /servers.[3]

The version of this document referred here 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, if you have the debian-policy installed, you can try FHS (local copy) (file:///usr/share/doc/debian-policy/fhs/)). The latest version, which may be a more recent version, may be found 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 the FHS mailing list (see the FHS web site (http://www.pathname.com/fhs/) for more information).

9.1.2 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 are not directories *in* /usr/local, but are children of directories in /usr/local. These directories (/usr/local/*/dir/) 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
    if chown root:staff /usr/local/share/emacs; then
      chmod 2775 /usr/local/share/emacs || true
    fi
  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.

9.1.3 The system-wide mail directory

The system-wide mail directory is /var/mail. This directory is part of the base system and should not be 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.

9.1.4 /run and /run/lock

The directory /run is cleared at boot, normally by being a mount point for a temporary file system. Packages therefore must not assume that any files or directories under /run other than /run/lock exist unless the package has arranged to create those files or directories since the last reboot. Normally, this is done by the package via an init script. See 'Writing the scripts' on page 62 for more information.

Packages must not include files or directories under /run, or under the older /var/run and /var/lock paths. The latter paths will normally be symlinks or other redirections to /run for backwards compatibility.

[3]These directories are used to store translators and as a set of standard names for mount points, respectively.

9.2 Users and groups

9.2.1 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`.

9.2.2 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`.

1000-59999: 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.

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.

9.3 System run levels and `init.d` scripts

9.3.1 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 behaviors 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`.

9.3.2 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 if it's already running, otherwise start 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 must ensure that they will behave sensibly (i.e., returning success and not starting multiple copies of a service) 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` with the `--oknodo` option.

Be careful of using `set -e` in `init.d` scripts. Writing correct `init.d` scripts requires accepting various error exit statuses when daemons are already running or already stopped without aborting the `init.d` script, and common `init.d` function libraries are not safe to call with `set -e` in effect[4]. For `init.d` scripts, it's often easier to not use `set -e` and instead check the result of each command separately.

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 must be treated as configuration files, either (if they are present in the package, that is, in the .deb file) by marking them as `conffiles`, or, (if they do not exist in the .deb) by managing them correctly in the maintainer scripts (see 'Configuration files' on page 74). 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

[4] `/lib/lsb/init-functions`, which assists in writing LSB-compliant init scripts, may fail if `set -e` is in effect and echoing status messages to the console fails, for example.

special command line options when starting a service, while making sure their 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 behavior of the scripts, and which a system administrator is likely to want to change. As the scripts themselves are frequently `conffile`s, 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 SUSv3 `sh` format. It may either be a `conffile` or a configuration file maintained by the package maintainer scripts. See 'Configuration files' on page 74 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.

Files and directories under `/run`, including ones referred to via the compatibility paths `/var/run` and `/var/lock`, are normally stored on a temporary filesystem and are normally not persistent across a reboot. The `init.d` scripts must handle this correctly. This will typically mean creating any required subdirectories dynamically when the `init.d` script is run. See '`/run` and `/run/lock`' on page 60 for more information.

9.3.3 Interfacing with the initscript system

Maintainers should use the abstraction layer provided by the `update-rc.d` and `invoke-rc.d` programs to deal with initscripts in their packages' scripts such as `postinst`, `prerm` and `postrm`.

Directly managing the /etc/rc?.d links and directly invoking the `/etc/init.d/` initscripts should be done only by packages providing the initscript subsystem (such as `sysv-rc` and `file-rc`).

Managing the links

The program `update-rc.d` is provided for package maintainers to arrange for the proper creation and removal of `/etc/rc`*`n`*`.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/rc`*`n`*`.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/rc`*`n`*`.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/rc`*`n`*`.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
```

and in your `postrm`

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

. Note that if your package changes runlevels or priority, you may have to remove and recreate the links, since otherwise the old links may persist. Refer to the documentation of `update-rc.d`.

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 man page `update-rc.d(8)`.

Running initscripts

The program `invoke-rc.d` is provided to make it easier for package maintainers to properly invoke an initscript, obeying runlevel and other locally-defined constraints that might limit a package's right to start, stop and otherwise manage services. This program may be used by maintainers in their packages' scripts.

The package maintainer scripts must use `invoke-rc.d` to invoke the `/etc/init.d/*` initscripts, instead of calling them directly.

By default, `invoke-rc.d` will pass any action requests (start, stop, reload, restart. . .) to the `/etc/init.d` script, filtering out requests to start or restart a service out of its intended runlevels.

Most packages will simply need to change:

```
/etc/init.d/<package>
       <action>
```

in their `postinst` and `prerm` scripts to:

```
 if which invoke-rc.d >/dev/null 2>&1; then
 invoke-rc.d package <action>
 else
 /etc/init.d/package <action>
 fi
```

A package should register its initscript services using `update-rc.d` before it tries to invoke them using `invoke-rc.d`. Invocation of unregistered services may fail.

For more information about using `invoke-rc.d`, please consult its man page `invoke-rc.d(8)`.

9.3.4 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 61. Packages must not place files in `/etc/rc.boot`.

9.3.5 Example

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

9.4 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 should be used for messages generated by `/etc/init.d` scripts.

- The 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 the script is performing some time consuming task in the background (not merely starting or stopping a program, for instance), an ellipsis (three dots: `...`) should be output to the screen, with no leading or tailing whitespace or line feeds.
- The messages should appear as if the computer is telling the user what it is doing (politely :-), but should not mention "it" directly. For example, instead of:

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

  the message should say

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

`init.d` script should use the following standard message formats for the situations enumerated below.

- When daemons are started

  If the 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 there are more than one daemon to start, the output should look like this:

  ```
  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 is happening and when the final daemon has been started. Care should be taken in the placement of white spaces: in the example above the system administrators can easily comment out a line if they don't want 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 same symbol (`"`) is used for the left and right quotation marks. A grave accent (`` ` ``) is not a quote character; neither is an apostrophe (`'`).

- 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 look 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 they have 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.

9.5 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 named as specified in 'Cron job file names' on this page into one or more of the following directories:

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

As these directory names imply, the files within them are executed on an hourly, 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 must be treated as configuration files.

If a certain job has to be executed at some other frequency or at a specific time, the package should install a file in `/etc/cron.d` with a name as specified in 'Cron job file names' on the current page. 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.)

Unlike `crontab` files described in the IEEE Std 1003.1-2008 (POSIX.1) available from The Open Group (`http://www.opengroup.org/onlinepubs/9699919799/`), the files in `/etc/cron.d` and the file `/etc/crontab` have seven fields; namely:

1 Minute [0,59]

2 Hour [0,23]

3 Day of the month [1,31]

4 Month of the year [1,12]

5 Day of the week ([0,6] with 0=Sunday)

6 Username

7 Command to be run

Ranges of numbers are allowed. Ranges are two numbers separated with a hyphen. The specified range is inclusive. Lists are allowed. A list is a set of numbers (or ranges) separated by commas. Step values can be used in conjunction with ranges.

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.

Any `cron` daemon must provide `/usr/bin/crontab` and support normal `crontab` entries as specified in POSIX. The daemon must also support names for days and months, ranges, and step values. It has to support `/etc/crontab`, and correctly execute the scripts in `/etc/cron.d`. The daemon must also correctly execute scripts in `/etc/cron.{hourly,daily,weekly,monthly}`.

9.5.1 Cron job file names

The file name of a cron job file should normally match the name of the package from which it comes.

If a package supplies multiple cron job files files in the same directory, the file names should all start with the name of the package (possibly modified as described below) followed by a hyphen (-) and a suitable suffix.

A cron job file name must not include any period or plus characters (`.` or +) characters as this will cause cron to ignore the file. Underscores (_) should be used instead of `.` and + characters.

9.6 Menus

The Debian `menu` package provides a standard interface between packages providing applications 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`.

Menu entries should follow the current menu policy.

The menu policy can be found in the `menu-policy` files in the `debian-policy` package. It is also available from the Debian web mirrors at `/doc/packaging-manuals/menu-policy/ (http://www.debian.org/doc/packaging-manuals/menu-policy/)`.

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

9.7 Multimedia handlers

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 invoke these handlers to view, edit or display MIME types they don't support directly.

Packages which provide programs to view/show/play, compose, edit or print MIME types should register them as such by placing a file in `mailcap(5)` format (RFC 1524) in the directory `/usr/lib/mime/packages/`. The file name should be the binary package's name.

The `mime-support` package provides the `update-mime` program, which integrates these registrations in the `/etc/mailcap` file, using dpkg triggers[5]. Packages using this facility *should not* depend on, recommend, or suggest `mime-support`.

9.8 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 ~`.

[5]Creating, modifying or removing a file in `/usr/lib/mime/packages/` using maintainer scripts will not activate the trigger. In that case, it can be done by calling `dpkg-trigger --no-await /usr/lib/mime/packages` from the maintainer script after creating, modifying, or removing the file.

- 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.

9.9 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.

9.10 Registering Documents using doc-base

The `doc-base` package implements a flexible mechanism for handling and presenting documentation. The recommended practice is for every Debian package that provides online documentation (other than just manual pages) to register these documents with `doc-base` by installing a `doc-base` control file in `/usr/share/doc-base/`.

Please refer to the documentation that comes with the `doc-base` package for information and details.

9.11 Alternate init systems

A number of other init systems are available now in Debian that can be used in place of `sysvinit`. Alternative init implementations must support running SysV init scripts as described at 'System run levels and `init.d` scripts' on page 61 for compatibility.

Packages may integrate with these replacement init systems by providing implementation-specific configuration information about how and when to start a service or in what order to run certain tasks at boot time. However, any package integrating with other init systems must also be backwards-compatible with `sysvinit` by providing a SysV-style init script with the same name as and equivalent functionality to any init-specific job, as this is the only start-up configuration method guaranteed to be supported by all init implementations. An exception to this rule is scripts or jobs provided by the init implementation itself; such jobs may be required for an implementation-specific equivalent of the `/etc/rcS.d/` scripts and may not have a one-to-one correspondence with the init scripts.

9.11.1 Event-based boot with upstart

Packages may integrate with the `upstart` event-based boot system by installing job files in the `/etc/init` directory. SysV init scripts for which an equivalent upstart job is available must query the output of the command `initctl version` for the string `upstart` and avoid running in favor of the native upstart job, using a test such as this:

```
if [ "$1" = start ] && which initctl >/dev/null && initctl version | grep -q upstart
then
 exit 1
fi
```

Because packages shipping upstart jobs may be installed on systems that are not using upstart, maintainer scripts must still use the common `update-rc.d` and `invoke-rc.d` interfaces for configuring runlevels and for starting and stopping services. These maintainer scripts must not call the upstart `start`, `restart`, `reload`, or `stop` interfaces directly. Instead, implementations of `invoke-rc.d` must detect when upstart is running and when an upstart job with the same name as an init script is present, and perform the requested action using the upstart job instead of the init script.

Dependency-based boot managers for SysV init scripts, such as `startpar`, may avoid running a given init script entirely when an equivalent upstart job is present, to avoid unnecessary forking of no-op init scripts. In this case, the boot manager should integrate with upstart to detect when the upstart job in question is started or stopped to know when the dependency has been satisfied.

Chapter 10

Files

10.1 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 12 and 'Conflicting binary packages - `Conflicts`' on page 44 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.

By default, when a package is being built, any binaries created should include debugging information, as well as being compiled with optimization. You should also turn on as many reasonable compilation warnings as possible; this makes life easier for porters, who can then look at build logs for possible problems. For the C programming language, this means the following compilation parameters should be used:

```
CC = gcc
CFLAGS = -O2 -g -Wall # sane warning options vary between programs
LDFLAGS = # none
INSTALL = 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.

Although binaries in the build tree should be compiled with debugging information by default, it can often be difficult to debug programs if they are also subjected to compiler optimization. For this reason, it is recommended to support the standardized environment variable `DEB_BUILD_OPTIONS` (see '`debian/rules` and `DEB_BUILD_OPTIONS`' on page 19). This variable can contain several flags to change how a package is compiled and built.

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.

10.2 Libraries

If the package is **architecture: any**, then the shared library compilation and linking flags must have `-fPIC`, or the package shall not build on some of the supported architectures[1]. Any exception to this rule must be discussed on the mailing list *debian-devel@lists.debian.org*, and a rough consensus obtained. The reasons for not compiling with `-fPIC` flag must be recorded in the file `README.Debian`, and care must be taken to either restrict the architecture or arrange for `-fPIC` to be used on architectures where it is required.[2]

As to the static libraries, the common case is not to have relocatable code, since there is no benefit, unless in specific cases; therefore the static version must not be compiled with the `-fPIC` flag. Any exception to this rule should be discussed on

[1]If you are using GCC, `-fPIC` produces code with relocatable position independent code, which is required for most architectures to create a shared library, with i386 and perhaps some others where non position independent code is permitted in a shared library. Position independent code may have a performance penalty, especially on `i386`. However, in most cases the speed penalty must be measured against the memory wasted on the few architectures where non position independent code is even possible.

[2]Some of the reasons why this might be required is if the library contains hand crafted assembly code that is not relocatable, the speed penalty is excessive for compute intensive libs, and similar reasons.

the mailing list *debian-devel@lists.debian.org*, and the reasons for compiling with the `-fPIC` flag must be recorded in the file `README.Debian`. [3]

In other words, if both a shared and a static library is being built, each source unit (`*.c`, for example, for C files) will need to be compiled twice, for the normal case.

Libraries should be built with threading support and to be thread-safe if the library supports this.

Although not enforced by the build tools, shared libraries must be linked against all libraries that they use symbols from in the same way that binaries are. This ensures the correct functioning of the symbols and shlibs systems and guarantees that all libraries can be safely opened with `dlopen()`. Packagers may wish to use the gcc option `-Wl,-z,defs` when building a shared library. Since this option enforces symbol resolution at build time, a missing library reference will be caught early as a fatal build error.
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 dynamic linking are in a separate part of the ELF object file.[4]

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.[5]

Packages that use `libtool` to create and install their shared libraries install a file containing additional metadata (ending in `.la`) alongside the library. For public libraries intended for use by other packages, these files normally should not be included in the Debian package, since the information they include is not necessary to link with the shared library on Debian and can add unnecessary additional dependencies to other programs or libraries.[6] If the `.la` file is required for that library (if, for instance, it's loaded via `libltdl` in a way that requires that meta-information), the `dependency_libs` setting in the `.la` file should normally be set to the empty string. If the shared library development package has historically included the `.la`, it must be retained in the development package (with `dependency_libs` emptied) until all libraries that depend on it have removed or emptied `dependency_libs` in their `.la` files to prevent linking with those other libraries using `libtool` from failing.

If the `.la` must be included, it should be included in the development (`-dev`) package, unless the library will be loaded by `libtool`'s `libltdl` library. If it is intended for use with `libltdl`, the `.la` files must go in the run-time library package.

These requirements for handling of `.la` files do not apply to loadable modules or libraries not installed in directories searched by default by the dynamic linker. Packages installing loadable modules will frequently need to install the `.la` files alongside the modules so that they can be loaded by `libltdl`. `dependency_libs` does not need to be modified for libraries or modules that are not installed in directories searched by the dynamic linker by default and not intended for use by other packages.

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.

10.3 Shared libraries

This section has moved to 'Shared libraries' on page 49.

[3] Some of the reasons for linking static libraries with the `-fPIC` flag are if, for example, one needs a Perl API for a library that is under rapid development, and has an unstable API, so shared libraries are pointless at this phase of the library's development. In that case, since Perl needs a library with relocatable code, it may make sense to create a static library with relocatable code. Another reason cited is if you are distilling various libraries into a common shared library, like `mklibs` does in the Debian installer project.

[4] 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.

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

[6] These files store, among other things, all libraries on which that shared library depends. Unfortunately, if the `.la` file is present and contains that dependency information, using `libtool` when linking against that library will cause the resulting program or library to be linked against those dependencies as well, even if this is unnecessary. This can create unneeded dependencies on shared library packages that would otherwise be hidden behind the library ABI, and can make library transitions to new SONAMEs unnecessarily complicated and difficult to manage.

10.4 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.

When scripts are installed into a directory in the system PATH, the script name should not include an extension such as .sh or .pl that denotes the scripting language currently used to implement it.

Shell scripts (sh and bash) other than init.d scripts should almost certainly start with set -e so that errors are detected. init.d scripts are something of a special case, due to how frequently they need to call commands that are allowed to fail, and it may instead be easier to check the exit status of commands directly. See 'Writing the scripts' on page 62 for more information about writing init.d scripts.

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

Scripts may assume that /bin/sh implements the SUSv3 Shell Command Language[7] plus the following additional features not mandated by SUSv3:[8]

- echo -n, if implemented as a shell built-in, must not generate a newline.
- test, if implemented as a shell built-in, must support -a and -o as binary logical operators.
- local to create a scoped variable must be supported, including listing multiple variables in a single local command and assigning a value to a variable at the same time as localizing it. local may or may not preserve the variable value from an outer scope if no assignment is present. Uses such as:

  ```
  fname () {
      local a b c=delta d
      # ... use a, b, c, d ...
  }
  ```

  must be supported and must set the value of c to delta.
- The XSI extension to kill allowing kill -*signal*, where *signal* is either the name of a signal or one of the numeric signals listed in the XSI extension (0, 1, 2, 3, 6, 9, 14, and 15), must be supported if kill is implemented as a shell built-in.
- The XSI extension to trap allowing numeric signals must be supported. In addition to the signal numbers listed in the extension, which are the same as for kill above, 13 (SIGPIPE) must be allowed.

If a shell script requires non-SUSv3 features from the shell interpreter other than those listed above, 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 SUSv3 features plus the above set when possible so that it may use /bin/sh as its interpreter. Checking your script with checkbashisms from the devscripts package or running your script with an alternate shell such as posh may help uncover violations of the above requirements. If in doubt whether a script complies with these requirements, 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://www.faqs.org/faqs/unix-faq/shell/csh-whynot/. 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 atomically 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.

[7]Single UNIX Specification, version 3, which is also IEEE 1003.1-2004 (POSIX), and is available on the World Wide Web from The Open Group (http://www.unix.org/version3/online.html) after free registration.

[8]These features are in widespread use in the Linux community and are implemented in all of bash, dash, and ksh, the most common shells users may wish to use as /bin/sh.

10.5 Symbolic links

In general, symbolic links within a top-level directory should be relative, and symbolic links pointing from one top-level directory to or into another should be absolute. (A top-level directory is a sub-directory of the root directory /.) For example, a symbolic link from /usr/lib/foo to /usr/share/bar should be relative (../share/bar), but a symbolic link from /var/run to /run should be absolute.[9]

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
```

A symbolic link pointing to a compressed file (in the sense that it is meant to be uncompressed with unzip or zless etc.) 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.)

10.6 Device files

Packages must not include device files or named pipes 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 notifying the user[10].

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*.

Named pipes needed by the package must be created in the postinst script[11] and removed in the prerm or postrm script as appropriate.

10.7 Configuration files

10.7.1 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 40).

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.

As noted elsewhere, /etc/init.d scripts, /etc/default files, scripts installed in /etc/cron.{hourly,daily,weekly,monthly}, and cron configuration installed in /etc/cron.d must be treated as configuration files. In general, any script that embeds configuration information is de-facto a configuration file and should be treated as such.

[9]This is necessary to allow top-level directories to be symlinks. If linking /var/run to /run were done with the relative symbolic link ../run, but /var were a symbolic link to /srv/disk1, the symbolic link would point to /srv/run rather than the intended target.

[10]This notification could be done via a (low-priority) debconf message, or an echo (printf) statement.

[11]It's better to use mkfifo rather than mknod to create named pipes so that automated checks for packages incorrectly creating device files with mknod won't have false positives.

10.7.2 Location

Any configuration files created or used by your package must reside in `/etc`. If there are several, 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 `/etc` directly, put the files in `/etc` and create symbolic links to those files from the location that the package requires.

10.7.3 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.

Obsolete configuration files without local changes should be removed by the package during upgrade.[12]

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.[13]

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 35 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 must 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.

10.7.4 Sharing configuration files

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.)

[12]The `dpkg-maintscript-helper` tool, available from the `dpkg` package, can help for this task.

[13]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 `conffile`s.

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.)

If the configuration file cannot be shared as described above, the packages must be marked as conflicting with each other. Two packages that specify the same file as a `conffile` must conflict. This is an instance of the general rule about not sharing files. Neither alternatives nor diversions are likely to be appropriate in this case; in particular, `dpkg` does not handle diverted `conffile`s well.

When two packages both declare the same `conffile`, they may see left-over configuration files from each other even though they conflict with each other. If a user removes (without purging) one of the packages and installs the other, the new package will take over the `conffile` from the old package. If the file was modified by the user, it will be treated the same as any other locally modified `conffile` during an upgrade.

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

10.7.5 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 are a bad thing, unless they do create the dotfiles themselves automatically.

Furthermore, programs should be configured by the Debian default installation to behave as closely to the upstream default behavior 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.

10.8 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 install a log rotation configuration file in the directory `/etc/logrotate.d`, normally named `/etc/logrotate.d/`*`package`*, and use the facilities provided by `logrotate`. [14] Here is a good example for a logrotate config file (for more information see `logrotate(8)`):

```
/var/log/foo/*.log {
    rotate 12
    weekly
    compress
    missingok
    postrotate
        start-stop-daemon -K -p /var/run/foo.pid -s HUP -x /usr/sbin/foo -q
    endscript
}
```

This rotates all files under `/var/log/foo`, saves 12 compressed generations, and tells the daemon to reopen its log files after the log rotation. It skips this log rotation (via `missingok`) if no such log file is present, which avoids errors if the package is removed but not purged.

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 40).

[14]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`).

10.9 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.

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.[15]

Control information files should be owned by `root:root` and either mode 644 (for most files) or mode 755 (for executables such as maintainer scripts).

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.[16] 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-passwd` 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).

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.

10.9.1 The use of `dpkg-statoverride`

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

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, they 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

[15]When a package is upgraded, and the owner or permissions of a file included in the package has changed, dpkg arranges for the ownership and permissions to be correctly set upon installation. However, this does not extend to directories; the permissions and ownership of directories already on the system does not change on install or upgrade of packages. This makes sense, since otherwise common directories like `/usr` would always be in flux. To correctly change permissions of a directory the package owns, explicit action is required, usually in the `postinst` script. Care must be taken to handle downgrades as well, in that case.

[16]Ordinary files installed by `dpkg` (as opposed to `conffile`s 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 behavior.

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. Care must be taken during upgrades to not override an existing setting.

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
  # only do something when no setting exists
  if ! dpkg-statoverride --list $i >/dev/null 2>&1
  then
    #include: debconf processing, question about foo and bar
    if [ "$RET" = "true" ] ; then
      dpkg-statoverride --update --add sysuser root 4755 $i
    fi
  fi
done
```

The corresponding code to remove the override when the package is purged would be:

```
for i in /usr/bin/foo /usr/sbin/bar
do
  if dpkg-statoverride --list $i >/dev/null 2>&1
  then
    dpkg-statoverride --remove $i
  fi
done
```

10.10 File names

The name of the files installed by binary packages in the system PATH (namely `/bin`, `/sbin`, `/usr/bin`, `/usr/sbin` and `/usr/games`) must be encoded in ASCII.

The name of the files and directories installed by binary packages outside the system PATH must be encoded in UTF-8 and should be restricted to ASCII when it is possible to do so.

Chapter 11

Customized programs

11.1 Architecture specification strings

If a program needs to specify an *architecture specification string* in some place, it should select one of the strings provided by `dpkg-architecture -L`. The strings are in the format *os*-*arch*, though the OS part is sometimes elided, as when the OS is Linux.

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.

11.1.1 Architecture wildcards

A package may specify an architecture wildcard. Architecture wildcards are in the format `any` (which matches every architecture), *os*-any, or any-*cpu*. [1]

11.2 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.

11.3 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 writable by group `utmp`. Programs which need to modify those files must be installed setgid `utmp`.

[1] Internally, the package system normalizes the GNU triplets and the Debian arches into Debian arch triplets (which are kind of inverted GNU triplets), with the first component of the triplet representing the libc and ABI in use, and then does matching against those triplets. However, such triplets are an internal implementation detail that should not be used by packages directly. The libc and ABI portion is handled internally by the package system based on the *os* and *cpu*.

11.4 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 their 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. Every package providing an editor or pager must call the `update-alternatives` script to register as an alternative for `/usr/bin/editor` or `/usr/bin/pager` as appropriate. The alternative should have a slave alternative for `/usr/share/man/man1/editor.1.gz` or `/usr/share/man/man1/pager.1.gz` pointing to the corresponding manual page.

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 `sensible-utils` package 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]

11.5 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.

1 Cgi-bin executable files are installed in the directory

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

or a subdirectory of that directory, and should be referred to as

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

(possibly with a subdirectory name before *cgi-bin-name*).

2 (Deleted)

3 Access to images

It is recommended that images for a package be stored in `/usr/share/images/`*`package`* and may be referred to through an alias `/images/` as

```
http://localhost/images/<package>/<filename>
```

4 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 `doc-base` 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.

5 Providing httpd and/or httpd-cgi

All web servers should provide the virtual package `httpd`. If a web server has CGI support it should provide `httpd-cgi` additionally.

All web applications which do not contain CGI scripts should depend on `httpd`, all those web applications which `do` contain CGI scripts, should depend on `httpd-cgi`.

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

11.6 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 either mode 600 and owned by *user* or mode 660 and owned by `user:mail`[5]. The local system administrator may choose a different permission scheme; packages should not make assumptions about the permission and ownership of mailboxes unless required (such as when creating a new mailbox). A MUA may remove a mailbox (unless it has nonstandard permissions) in which case the MTA or another MUA must recreate it if needed.

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 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 a 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`.

11.7 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

[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.

[5]There are two traditional permission schemes for mail spools: mode 600 with all mail delivery done by processes running as the destination user, or mode 660 and owned by group mail with mail delivery done by a process running as a system user in group mail. Historically, Debian required mode 660 mail spools to enable the latter model, but that model has become increasingly uncommon and the principle of least privilege indicates that mail systems that use the first model should use permissions of 600. If delivery to programs is permitted, it's easier to keep the mail system secure if the delivery agent runs as the destination user. Debian Policy therefore permits either scheme.

`/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.

11.8 Programs for the X Window System

11.8.1 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.

11.8.2 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 `Provides` control field that they provide the virtual package `xserver`.[6]

11.8.3 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 `Provides` control field 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. That alternative should have a slave alternative for `/usr/share/man/man1/x-terminal-emulator.1.gz` pointing to the corresponding manual page.

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

- 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[7] and runs the specified *command*, interpreting the entirety of the rest of the command line as a command to pass straight to exec, in the manner that `xterm` does.
- Support the command-line option `-T` *`title`*, which creates a new terminal window with the window title *title*.

11.8.4 Packages providing a window manager

Packages that provide a window manager should declare in their `Provides` control field 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 complies with The Window Manager Specification Project (`http://www.freedesktop.org/wiki/Specifications/wm-spec`), written by the Free Desktop Group (`http://www.freedesktop.org/wiki/`), add 40 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.

That alternative should have a slave alternative for `/usr/share/man/man1/x-window-manager.1.gz` pointing to the corresponding manual page.

[6] 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.

[7] "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).

11.8.5 Packages providing fonts

Packages that provide fonts for the X Window System[8] 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 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.[9]

2 BDF fonts must be converted to PCF fonts with the `bdftopcf` utility (available in the `xfonts-utils` package, `gzip`ped, and placed in a directory that corresponds to their resolution:
   - 100 dpi fonts must be placed in `/usr/share/fonts/X11/100dpi/`.
   - 75 dpi fonts must be placed in `/usr/share/fonts/X11/75dpi/`.
   - Character-cell fonts, cursor fonts, and other low-resolution fonts must be placed in `/usr/share/fonts/X11/misc/`.

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

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

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

6 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.

7 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.

8 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/`*`fontdir`*`/`*`package`*`.`*`extension`*, where *fontdir* is the name of the subdirectory of `/usr/share/fonts/X11/` 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.

9 Font packages must declare a dependency on `xfonts-utils` in their `Depends` or `Pre-Depends` control field.

10 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.

11 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.

12 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.

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

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

[8]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 renderer, 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.

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

11.8.6 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 `conffile`s or handled as configuration files.

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 be registered as a `conffile` or handled as a configuration file.[10]

11.8.7 Installation directory issues

Historically, packages using the X Window System used a separate set of installation directories from other packages. This practice has been discontinued and packages using the X Window System should now generally be installed in the same directories as any other package. Specifically, packages must not install files under the `/usr/X11R6/` directory and the `/usr/X11R6/` directory hierarchy should be regarded as obsolete.

Include files previously installed under `/usr/X11R6/include/X11/` should be installed into `/usr/include/X11/`. For files previously installed into subdirectories of `/usr/X11R6/lib/X11/`, package maintainers should determine if subdirectories of `/usr/lib/` and `/usr/share/` can be used. If not, a subdirectory of `/usr/lib/X11/` should be used.

Configuration files for window, display, or session managers or other applications that are tightly integrated with the X Window System may be placed in a subdirectory of `/etc/X11/` corresponding to the package name. Other X Window System applications should use the `/etc/` directory unless otherwise mandated by policy (such as for 'Application defaults files' on the current page).

11.9 Perl programs and modules

Perl programs and modules should follow the current Perl policy.

The Perl policy can be found in the `perl-policy` files in the `debian-policy` package. It is also available from the Debian web mirrors at `/doc/packaging-manuals/perl-policy/` (`http://www.debian.org/doc/packaging-manuals/perl-policy/`).

11.10 Emacs lisp programs

Please refer to the "Debian Emacs Policy" for details of how to package emacs lisp programs.

The Emacs policy is available in `debian-emacs-policy.gz` of the `emacsen-common` package. It is also available from the Debian web mirrors at `/doc/packaging-manuals/debian-emacs-policy` (`http://www.debian.org/doc/packaging-manuals/debian-emacs-policy`).

11.11 Games

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

Each game decides on its own security policy.

Games which require protected, privileged access to high-score files, saved games, 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

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

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 12

Documentation

12.1 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 pre-formatted “cat page”.

Each program, utility, and function should have an associated manual page included in the same package. It is suggested that all configuration files also have a manual page included as well. Manual pages for protocols and other auxiliary things are optional.

If no manual page is available, this is considered as a bug and should be reported to the Debian Bug Tracking System (the maintainer of the package is allowed to write this bug report themselves, if they so desire). Do not close the bug report until a proper man page is available.[1]

You may forward a complaint about a missing man page 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 man page 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 man page 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 man page should be relative to the base of the man page tree (usually /usr/share/man). If you do not create any links (whether symlinks, hard links, or .so directives) in the file system to the alternate names of the man page, then you should not rely on man finding your man page under those names based solely on the information in the man page’s header.[2]

Manual pages in locale-specific subdirectories of /usr/share/man should use either UTF-8 or the usual legacy encoding for that language (normally the one corresponding to the shortest relevant locale name in /usr/share/i18n /SUPPORTED). For example, pages under /usr/share/man/fr should use either UTF-8 or ISO-8859-1.[3]

A country name (the DE in de_DE) should not be included in the subdirectory name unless it indicates a significant difference in the language, as this excludes speakers of the language in other countries.[4]

If a localized version of a manual page is provided, it should either be up-to-date or it should be obvious to the reader that it is outdated and the original manual page should be used instead. This can be done either by a note at the beginning of the manual page or by showing the missing or changed portions in the original language instead of the target language.

12.2 Info documents

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

[1]It is not very hard to write a man page. See the Man-Page-HOWTO (http://www.schweikhardt.net/man_page_howto.html), man(7), the examples created by dh_make, the helper program help2man, or the directory /usr/share/doc/man-db/examples.

[2]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 file system. This support is therefore deprecated and will cease to be present in the future.

[3]man will automatically detect whether UTF-8 is in use. In future, all manual pages will be required to use UTF-8.

[4]At the time of writing, Chinese and Portuguese are the main languages with such differences, so pt_BR, zh_CN, and zh_TW are all allowed.

The install-info program maintains a directory of installed info documents in /usr/share/info/dir for the use of info readers. This file must not be included in packages other than install-info.

install-info is automatically invoked when appropriate using dpkg triggers. Packages other than install-info *should not* invoke install-info directly and *should not* depend on, recommend, or suggest install-info for this purpose.

Info readers requiring the /usr/share/info/dir file should depend on install-info.

Info documents should contain section and directory entry information in the document for the use of install-info. The section should be specified via a line starting with INFO-DIR-SECTION followed by a space and the section of this info page. The directory entry or entries should be included between a START-INFO-DIR-ENTRY line and an END-INFO-DIR-ENTRY line. For example:

```
INFO-DIR-SECTION Individual utilities
START-INFO-DIR-ENTRY
* example: (example).              An example info directory entry.
END-INFO-DIR-ENTRY
```

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).[5]

12.3 Additional documentation

Any additional documentation that comes with the package may be installed at the discretion of the package maintainer. Plain 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.

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!

Packages must not require the existence of any files in /usr/share/doc/ in order to function [6]. Any files that are referenced by programs but are also useful as stand alone documentation should be installed under /usr/share/*package*/ with symbolic links from /usr/share/doc/*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.[7]

Former Debian releases placed all additional documentation in /usr/doc/*package*. This has been changed to /usr/share/doc/*package*, and packages must not put documentation in the directory /usr/doc/*package*. [8]

12.4 Preferred documentation formats

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

If your package comes with extensive documentation in a markup format that can be converted to various other formats you should if possible ship HTML versions in a binary package, in the directory /usr/share/doc/*appropriate-package* or its subdirectories.[9]

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

[5] Normally, info documents are generated from Texinfo source. To include this information in the generated info document, if it is absent, add commands like:

```
@dircategory Individual utilities @direntry * example: (example). An example info directory entry. @end direntry
```

to the Texinfo source of the document and ensure that the info documents are rebuilt from source during the package build.

[6] The system administrator should be able to delete files in /usr/share/doc/ without causing any programs to break.

[7] Please note that this does not override the section on changelog files below, so the file /usr/share/doc/*package*/changelog.Debian.gz must refer to the changelog for the current version of *package* in question. In practice, this means that the sources of the target and the destination of the symlink must be the same (same source package and version).

[8] At this phase of the transition, we no longer require a symbolic link in /usr/doc/. At a later point, policy shall change to make the symbolic links a bug.

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

12.5 Copyright information

Every package must be accompanied by a verbatim copy of its copyright information 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 name the original authors.

Packages in the *contrib* or *non-free* archive areas should state in the copyright file that the package is not part of the Debian distribution and briefly explain why.

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 `copyright` files must be extractable by mechanical means.

Packages distributed under the Apache license (version 2.0), the Artistic license, the GNU GPL (versions 1, 2, or 3), the GNU LGPL (versions 2, 2.1, or 3), and the GNU FDL (versions 1.2 or 1.3) should refer to the corresponding files under `/usr/share/common-licenses`,[10] 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.

All copyright files must be encoded in UTF-8.

12.5.1 Machine-readable copyright information

A specification for a standard, machine-readable format for `debian/copyright` files is maintained as part of the `debian-policy` package. This document may be found in the `copyright-format` files in the `debian-policy` package. It is also available from the Debian web mirrors at `/doc/packaging-manuals/copyright-format/1.0/` (`http://www.debian.org/doc/packaging-manuals/copyright-format/1.0/`).

Use of this format is optional.

12.6 Examples

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

If the purpose of a package is to provide examples, then the example files may be installed into `/usr/share/doc/`*`package`*.

12.7 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`

[10] In particular, `/usr/share/common-licenses/Apache-2.0`, `/usr/share/common-licenses/Artistic`, `/usr/share/common-licenses/GPL-1`, `/usr/share/common-licenses/GPL-2`, `/usr/share/common-licenses/GPL-3`, `/usr/share/common-licenses/LGPL-2`, `/usr/share/common-licenses/LGPL-2.1`, `/usr/share/common-licenses/LGPL-3`, `/usr/share/common-licenses/GFDL-1.2`, and `/usr/share/common-licenses/GFDL-1.3` respectively. The University of California BSD license is also included in `base-files` as `/usr/share/common-licenses/BSD`, but given the brevity of this license, its specificity to code whose copyright is held by the Regents of the University of California, and the frequency of minor wording changes, its text should be included in the copyright file rather than referencing this file.

`-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.[11]

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`.

For details about the format and contents of the Debian changelog file, please see 'Debian changelog: `debian/changelog`' on page 16.

[11]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.

Certain parts of the Packaging manual were integrated into the Policy Manual proper, and removed from the appendices. Links have been placed from the old locations to the new ones.

`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 behavior of the package management programs `dpkg`, `dselect` et al. and the way they interact with packages.

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 conjunction with those programs' man pages.

The utility programs which are provided with `dpkg` not described in detail here, are documented in their man pages.

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. However, while the examples are helpful, they do not replace the need to read and follow the Policy and Programmer's Manual.

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

Appendix B

Binary packages (from old Packaging Manual)

See `deb(5)` and 'Package control information files' on the current page.

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 file system 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 file system 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 the following page).

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

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 man page `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 - --wildcards \*/copyright | pager
```

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 information file area, but this is not generally a good idea (though they will largely be ignored).

Here is a brief list of the control information 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 'Source package control files – `debian/control`' on page 24 and 'Binary package control files – `DEBIAN/control`' on page 24.

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 the next page.

`postinst`, `preinst`, `postrm`, `prerm` These are executable 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 35.

It is very important to make these scripts idempotent. See 'Maintainer scripts idempotency' on page 35.

The maintainer scripts are not guaranteed to run with a controlling terminal and may not be able to interact with the user. See 'Controlling terminal for maintainer scripts' on page 36.

`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 101). 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 56.

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 the next page for more details.

The fields in binary package control files are listed in 'Binary package control files – `DEBIAN/control`' on page 24.

A description of the syntax of control files and the purpose of the fields is available in 'Control files and their fields' on page 23.

B.4 Time Stamps

See 'Time Stamps' on page 17.

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.

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 97.

C.1.2 `dpkg-buildpackage` - overall package-building control script

See `dpkg-buildpackage(1)`.

C.1.3 `dpkg-gencontrol` - generates binary package control files

This program is usually called from `debian/rules` (see 'The Debian package source tree' on page 97) 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.

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

See `dpkg-shlibdeps(1)`.

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.

C.1.6 `dpkg-genchanges` - generates a `.changes` upload control file

See `dpkg-genchanges(1)`.

C.1.7 `dpkg-parsechangelog` - produces parsed representation of a changelog

See `dpkg-parsechangelog(1)`.

C.1.8 `dpkg-architecture` - information about the build and host system

See `dpkg-architecture(1)`.

[1] This is so that the control file which is produced has the right permissions

C.2 The Debian package source tree

The source archive scheme described later is intended to allow a Debian package source tree with some associated control information to be reproduced and transported easily. The Debian package source tree is a version of the original program with certain files added for the benefit of the packaging 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 Debian package source tree. They are described below.

C.2.1 `debian/rules` - the main building script

See 'Main building script: `debian/rules`' on page 18.

C.2.2 `debian/substvars` and variable substitutions

See 'Variable substitutions: `debian/substvars`' on page 20.

C.2.3 `debian/files`

See 'Generated files list: `debian/files`' on page 21.

C.2.4 `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 file system 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 93.

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 is a control file used by `dpkg-source` to extract a source package. See 'Debian source control files – `.dsc`' on page 25.

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.

Debian package 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 preferably contains a directory named *`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 Debian 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 [2] [3], device special files, sockets or setuid or setgid files. [4]

The source packaging tools manage the changes between the original and Debian source using `diff` and `patch`. Turning the original source tree as included in the `.orig.tar.gz` into the Debian package 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. [5]
- 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-executable.

[2]This is not currently detected when building source packages, but only when extracting them.

[3]Hard links may be permitted at some point in the future, but would require a fair amount of work.

[4]Setgid directories are allowed.

[5]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

See 'Syntax of control files' on page 23.

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.

D.2 List of fields

See 'List of fields' on page 26.

This section now contains only the fields that didn't belong to the Policy manual.

D.2.1 `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.2.2 `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.2.3 `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 re-installation) or not and what its current state on the system is. Each of these pieces of information is a single word.

D.2.4 `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.2.5 `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.2.6 Obsolete fields

These are still recognized 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 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 information 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 file system.

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 man page `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 103 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:

```
dpkg-divert --package smailwrapper --add --rename \
   --divert /usr/sbin/smail.real /usr/sbin/smail
```

The --package smailwrapper ensures that smailwrapper's copy of /usr/sbin/smail can bypass the diversion and get installed as the true version. It's safe to add the diversion unconditionally on upgrades since it will be left unchanged if it already exists, but dpkg-divert will display a message. To suppress that message, make the command conditional on the version from which the package is being upgraded:

```
if [ upgrade != "$1" ] || dpkg --compare-versions "$2" lt 1.0-2; then
   dpkg-divert --package smailwrapper --add --rename \
      --divert /usr/sbin/smail.real /usr/sbin/smail
fi
```

where 1.0-2 is the version at which the diversion was first added to the package. Running the command during abort-upgrade is pointless but harmless.

The postrm has to do the reverse:

```
if [ remove = "$1" -o abort-install = "$1" -o disappear = "$1" ]; then
   dpkg-divert --package smailwrapper --remove --rename \
      --divert /usr/sbin/smail.real /usr/sbin/smail
fi
```

If the diversion was added at a particular version, the postrm should also handle the failure case of upgrading from an older version (unless the older version is so old that direct upgrades are no longer supported):

```
if [ abort-upgrade = "$1" ] && dpkg --compare-versions "$2" lt 1.0-2; then
   dpkg-divert --package smailwrapper --remove --rename \
      --divert /usr/sbin/smail.real /usr/sbin/smail
fi
```

where `1.0-2` is the version at which the diversion was first added to the package. The postrm should not remove the diversion on upgrades both because there's no reason to remove the diversion only to immediately re-add it and since the postrm of the old package is run after unpacking so the removal of the diversion will fail.

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.

Do not attempt to divert a conffile, as `dpkg` does not handle it well.