Debian Developer's Reference

Developer's Reference Team
<developers-reference@packages.debian.org>
Andreas Barth
Adam Di Carlo
Raphaël Hertzog
Christian Schwarz
Ian Jackson

ver. 3.3.9, 04 August, 2007

Copyright Notice

Contents

Chapter 1

Scope of This Document

The purpose of this document is to provide an overview of the recommended procedures and the available resources for Debian developers.

The procedures discussed within include how to become a maintainer (‘Applying to Become a Maintainer’ on page 3); how to create new packages (‘New packages’ on page 31) and how to upload packages (‘Uploading a package’ on page 35); how to handle bug reports (‘Handling bugs’ on page 37); how to move, remove, or orphan packages (‘Moving, removing, renaming, adopting, and orphaning packages’ on page 45); how to port packages (‘Porting and being ported’ on page 48); and how and when to do interim releases of other maintainers’ packages (‘Non-Maintainer Uploads (NMUs)’ on page 53).

The resources discussed in this reference include the mailing lists (‘Mailing lists’ on page 11) and servers (‘Debian machines’ on page 14); a discussion of the structure of the Debian archive (‘The Debian archive’ on page 17); explanation of the different servers which accept package uploads (‘Uploading to `ftp-master`’ on page 35); and a discussion of resources which can help maintainers with the quality of their packages (‘Overview of Debian Maintainer Tools’ on page 99).

It should be clear that this reference does not discuss the technical details of Debian packages nor how to generate them. Nor does this reference detail the standards to which Debian software must comply. All of such information can be found in the Debian Policy Manual (`http://www.debian.org/doc/debian-policy/`).

Furthermore, this document is *not an expression of formal policy*. It contains documentation for the Debian system and generally agreed-upon best practices. Thus, it is not what is called a “normative” document.

Chapter 2

Applying to Become a Maintainer

2.1 Getting started

So, you've read all the documentation, you've gone through the Debian New Maintainers' Guide (http://www.debian.org/doc/maint-guide/), understand what everything in the `hello` example package is for, and you're about to Debianize your favorite piece of software. How do you actually become a Debian developer so that your work can be incorporated into the Project?

Firstly, subscribe to <debian-devel@lists.debian.org> if you haven't already. Send the word `subscribe` in the *Subject* of an email to <debian-devel-REQUEST@lists.debian.org>. In case of problems, contact the list administrator at <listmaster@lists.debian.org>. More information on available mailing lists can be found in 'Mailing lists' on page 11. <debian-devel-announce@lists.debian.org> is another list which is mandatory for anyone who wishes to follow Debian's development.

You should subscribe and lurk (that is, read without posting) for a bit before doing any coding, and you should post about your intentions to work on something to avoid duplicated effort.

Another good list to subscribe to is <debian-mentors@lists.debian.org>. See 'Debian mentors and sponsors' on the next page for details. The IRC channel `#debian` can also be helpful; see 'IRC channels' on page 13.

When you know how you want to contribute to Debian GNU/Linux, you should get in contact with existing Debian maintainers who are working on similar tasks. That way, you can learn from experienced developers. For example, if you are interested in packaging existing software for Debian, you should try to get a sponsor. A sponsor will work together with you on your package and upload it to the Debian archive once they are happy with the packaging work you have done. You can find a sponsor by mailing the <debian-mentors@lists.debian.org> mailing list, describing your package and yourself and asking for a sponsor (see 'Sponsoring packages' on page 93 and http://people.debian.org/~mpalmer/debian-mentors_FAQ.html for more information on sponsoring). On the other hand, if you are interested in porting Debian to alternative architectures or kernels you can subscribe to port specific mailing lists and ask there how to get started. Finally, if you are interested in

documentation or Quality Assurance (QA) work you can join maintainers already working on these tasks and submit patches and improvements.

One pitfall could be a too-generic local part in your mailadress: Terms like mail, admin, root, master should be avoided, please see http://www.debian.org/MailingLists/ for details.

2.2 Debian mentors and sponsors

The mailing list <debian-mentors@lists.debian.org> has been set up for novice maintainers who seek help with initial packaging and other developer-related issues. Every new developer is invited to subscribe to that list (see 'Mailing lists' on page 11 for details).

Those who prefer one-on-one help (e.g., via private email) should also post to that list and an experienced developer will volunteer to help.

In addition, if you have some packages ready for inclusion in Debian, but are waiting for your new maintainer application to go through, you might be able find a sponsor to upload your package for you. Sponsors are people who are official Debian Developers, and who are willing to criticize and upload your packages for you. Please read the unofficial debian-mentors FAQ at http://people.debian.org/~mpalmer/debian-mentors_FAQ.html first.

If you wish to be a mentor and/or sponsor, more information is available in 'Interacting with prospective Debian developers' on page 93.

2.3 Registering as a Debian developer

Before you decide to register with Debian GNU/Linux, you will need to read all the information available at the New Maintainer's Corner (http://www.debian.org/devel/join/newmaint). It describes in detail the preparations you have to do before you can register to become a Debian developer. For example, before you apply, you have to read the Debian Social Contract (http://www.debian.org/social_contract). Registering as a developer means that you agree with and pledge to uphold the Debian Social Contract; it is very important that maintainers are in accord with the essential ideas behind Debian GNU/Linux. Reading the GNU Manifesto (http://www.gnu.org/gnu/manifesto.html) would also be a good idea.

The process of registering as a developer is a process of verifying your identity and intentions, and checking your technical skills. As the number of people working on Debian GNU/Linux has grown to over 900 and our systems are used in several very important places, we have to be careful about being compromised. Therefore, we need to verify new maintainers before we can give them accounts on our servers and let them upload packages.

Before you actually register you should have shown that you can do competent work and will be a good contributor. You show this by submitting patches through the Bug Tracking System and having a package sponsored by an existing Debian Developer for a while. Also,

we expect that contributors are interested in the whole project and not just in maintaining their own packages. If you can help other maintainers by providing further information on a bug or even a patch, then do so!

Registration requires that you are familiar with Debian's philosophy and technical documentation. Furthermore, you need a GnuPG key which has been signed by an existing Debian maintainer. If your GnuPG key is not signed yet, you should try to meet a Debian Developer in person to get your key signed. There's a GnuPG Key Signing Coordination page (`http://nm.debian.org/gpg.php`) which should help you find a Debian Developer close to you. (If there is no Debian Developer close to you, alternative ways to pass the ID check may be permitted as an absolute exception on a case-by-case-basis. See the identification page (`http://www.debian.org/devel/join/nm-step2`) for more information.)

If you do not have an OpenPGP key yet, generate one. Every developer needs an OpenPGP key in order to sign and verify package uploads. You should read the manual for the software you are using, since it has much important information which is critical to its security. Many more security failures are due to human error than to software failure or high-powered spy techniques. See 'Maintaining your public key' on page 7 for more information on maintaining your public key.

Debian uses the `GNU Privacy Guard` (package `gnupg` version 1 or better) as its baseline standard. You can use some other implementation of OpenPGP as well. Note that OpenPGP is an open standard based on RFC 2440 (`http://www.rfc-editor.org/rfc/rfc2440.txt`).

You need a version 4 key for use in Debian Development. Your key length must be at least 1024 bits; there is no reason to use a smaller key, and doing so would be much less secure. [1]

If your public key isn't on a public key server such as `subkeys.pgp.net`, please read the documentation available at NM Step 2: Identification (`http://www.debian.org/devel/join/nm-step2`). That document contains instructions on how to put your key on the public key servers. The New Maintainer Group will put your public key on the servers if it isn't already there.

Some countries restrict the use of cryptographic software by their citizens. This need not impede one's activities as a Debian package maintainer however, as it may be perfectly legal to use cryptographic products for authentication, rather than encryption purposes. If you live in

[1] Version 4 keys are keys conforming to the OpenPGP standard as defined in RFC 2440. Version 4 is the key type that has always been created when using GnuPG. PGP versions since 5.x also could create v4 keys, the other choice having beein pgp 2.6.x compatible v3 keys (also called "legacy RSA" by PGP). Version 4 (primary) keys can either use the RSA or the DSA algorithms, so this has nothing to do with GnuPG's question about "which kind of key do you want: (1) DSA and Elgamal, (2) DSA (sign only), (5) RSA (sign only)". If you don't have any special requirements just pick the default. The easiest way to tell whether an existing key is a v4 key or a v3 (or v2) key is to look at the fingerprint: Fingerprints of version 4 keys are the SHA-1 hash of some key matieral, so they are 40 hex digits, usually grouped in blocks of 4. Fingerprints of older key format versions used MD5 and are generally shown in blocks of 2 hex digits. For example if your fingerprint looks like `5B00 C96D 5D54 AEE1 206B  AF84 DE7A AF6E 94C0 9C7F` then it's a v4 key. Another possibility is to pipe the key into `pgpdump`, which will say something like "Public Key Packet - Ver 4". Also note that your key must be self-signed (i.e. it has to sign all its own user IDs; this prevents user ID tampering). All modern OpenPGP software does that automatically, but if you have an older key you may have to manually add those signatures.

a country where use of cryptography even for authentication is forbidden then please contact us so we can make special arrangements.

To apply as a new maintainer, you need an existing Debian Developer to support your application (an *advocate*). After you have contributed to Debian for a while, and you want to apply to become a registered developer, an existing developer with whom you have worked over the past months has to express their belief that you can contribute to Debian successfully.

When you have found an advocate, have your GnuPG key signed and have already contributed to Debian for a while, you're ready to apply. You can simply register on our application page (http://nm.debian.org/newnm.php). After you have signed up, your advocate has to confirm your application. When your advocate has completed this step you will be assigned an Application Manager who will go with you through the necessary steps of the New Maintainer process. You can always check your status on the applications status board (http://nm.debian.org/).

For more details, please consult New Maintainer's Corner (http://www.debian.org/devel/join/newmaint) at the Debian web site. Make sure that you are familiar with the necessary steps of the New Maintainer process before actually applying. If you are well prepared, you can save a lot of time later on.

Chapter 3

Debian Developer's Duties

3.1 Maintaining your Debian information

There's a LDAP database containing information about Debian developers at https://db.debian.org/. You should enter your information there and update it as it changes. Most notably, make sure that the address where your debian.org email gets forwarded to is always up to date, as well as the address where you get your debian-private subscription if you choose to subscribe there.

For more information about the database, please see 'The Developers Database' on page 16.

3.2 Maintaining your public key

Be very careful with your private keys. Do not place them on any public servers or multiuser machines, such as the Debian servers (see 'Debian machines' on page 14). Back your keys up; keep a copy offline. Read the documentation that comes with your software; read the PGP FAQ (http://www.cam.ac.uk.pgp.net/pgpnet/pgp-faq/).

You need to ensure not only that your key is secure against being stolen, but also that it is secure against being lost. Generate and make a copy (best also in paper form) of your revocation certificate; this is needed if your key is lost.

If you add signatures to your public key, or add user identities, you can update the Debian key ring by sending your key to the key server at `keyring.debian.org`.

If you need to add a completely new key or remove an old key, you need to get the new key signed by another developer. If the old key is compromised or invalid, you also have to add the revocation certificate. If there is no real reason for a new key, the Keyring Maintainers might reject the new key. Details can be found at http://keyring.debian.org/replacing_keys.html.

The same key extraction routines discussed in 'Registering as a Debian developer' on page 4 apply.

You can find a more in-depth discussion of Debian key maintenance in the documentation of the `debian-keyring` package.

3.3 Voting

Even though Debian isn't really a democracy, we use a democratic process to elect our leaders and to approve general resolutions. These procedures are defined by the Debian Constitution (`http://www.debian.org/devel/constitution`).

Other than the yearly leader election, votes are not routinely held, and they are not undertaken lightly. Each proposal is first discussed on the `<debian-vote@lists.debian.org>` mailing list and it requires several endorsements before the project secretary starts the voting procedure.

You don't have to track the pre-vote discussions, as the secretary will issue several calls for votes on `<debian-devel-announce@lists.debian.org>` (and all developers are expected to be subscribed to that list). Democracy doesn't work well if people don't take part in the vote, which is why we encourage all developers to vote. Voting is conducted via GPG-signed/encrypted email messages.

The list of all proposals (past and current) is available on the Debian Voting Information (`http://www.debian.org/vote/`) page, along with information on how to make, second and vote on proposals.

3.4 Going on vacation gracefully

It is common for developers to have periods of absence, whether those are planned vacations or simply being buried in other work. The important thing to notice is that other developers need to know that you're on vacation so that they can do whatever is needed if a problem occurs with your packages or other duties in the project.

Usually this means that other developers are allowed to NMU (see 'Non-Maintainer Uploads (NMUs)' on page 53) your package if a big problem (release critical bug, security update, etc.) occurs while you're on vacation. Sometimes it's nothing as critical as that, but it's still appropriate to let others know that you're unavailable.

In order to inform the other developers, there are two things that you should do. First send a mail to `<debian-private@lists.debian.org>` with "[VAC] " prepended to the subject of your message[1] and state the period of time when you will be on vacation. You can also give some special instructions on what to do if a problem occurs.

The other thing to do is to mark yourself as "on vacation" in the Debian developers' LDAP database (this information is only accessible to Debian developers). Don't forget to remove the "on vacation" flag when you come back!

[1] This is so that the message can be easily filtered by people who don't want to read vacation notices.

Ideally, you should sign up at the GPG coordination site (http://nm.debian.org/gpg.php) when booking a holiday and check if anyone there is looking for signing. This is especially important when people go to exotic places where we don't have any developers yet but where there are people who are interested in applying.

3.5 Coordination with upstream developers

A big part of your job as Debian maintainer will be to stay in contact with the upstream developers. Debian users will sometimes report bugs that are not specific to Debian to our bug tracking system. You have to forward these bug reports to the upstream developers so that they can be fixed in a future upstream release.

While it's not your job to fix non-Debian specific bugs, you may freely do so if you're able. When you make such fixes, be sure to pass them on to the upstream maintainers as well. Debian users and developers will sometimes submit patches to fix upstream bugs — you should evaluate and forward these patches upstream.

If you need to modify the upstream sources in order to build a policy compliant package, then you should propose a nice fix to the upstream developers which can be included there, so that you won't have to modify the sources of the next upstream version. Whatever changes you need, always try not to fork from the upstream sources.

3.6 Managing release-critical bugs

Generally you should deal with bug reports on your packages as described in 'Handling bugs' on page 37. However, there's a special category of bugs that you need to take care of — the so-called release-critical bugs (RC bugs). All bug reports that have severity *critical*, *grave* or *serious* are considered to have an impact on whether the package can be released in the next stable release of Debian. These bugs can delay the Debian release and/or can justify the removal of a package at freeze time. That's why these bugs need to be corrected as quickly as possible.

Developers who are part of the Quality Assurance (http://qa.debian.org/) group are following all such bugs, and trying to help whenever possible. If, for any reason, you aren't able fix an RC bug in a package of yours within 2 weeks, you should either ask for help by sending a mail to the Quality Assurance (QA) group <debian-qa@lists.debian.org>, or explain your difficulties and present a plan to fix them by sending a mail to the bug report. Otherwise, people from the QA group may want to do a Non-Maintainer Upload (see 'Non-Maintainer Uploads (NMUs)' on page 53) after trying to contact you (they might not wait as long as usual before they do their NMU if they have seen no recent activity from you in the BTS).

3.7 Retiring

If you choose to leave the Debian project, you should make sure you do the following steps:

1 Orphan all your packages, as described in 'Orphaning a package' on page 47.

2 Send an gpg-signed email about why you are leaving the project to <debian-private@lists.debian.org>.

3 Notify the Debian key ring maintainers that you are leaving by opening a ticket in Debian RT by sending a mail to keyring@rt.debian.org with the words 'Debian RT' somewhere in the subject line (case doesn't matter).

Chapter 4

Resources for Debian Developers

In this chapter you will find a very brief road map of the Debian mailing lists, the Debian machines which may be available to you as a developer, and all the other resources that are available to help you in your maintainer work.

4.1 Mailing lists

Much of the conversation between Debian developers (and users) is managed through a wide array of mailing lists we host at `lists.debian.org` (`http://lists.debian.org/`). To find out more on how to subscribe or unsubscribe, how to post and how not to post, where to find old posts and how to search them, how to contact the list maintainers and see various other information about the mailing lists, please read `http://www.debian.org/MailingLists/`. This section will only cover aspects of mailing lists that are of particular interest to developers.

4.1.1 Basic rules for use

When replying to messages on the mailing list, please do not send a carbon copy (`CC`) to the original poster unless they explicitly request to be copied. Anyone who posts to a mailing list should read it to see the responses.

Cross-posting (sending the same message to multiple lists) is discouraged. As ever on the net, please trim down the quoting of articles you're replying to. In general, please adhere to the usual conventions for posting messages.

Please read the code of conduct (`http://www.debian.org/MailingLists/#codeofconduct`) for more information. The Debian Community Guidelines (`http://people.debian.org/~enrico/dcg/`) are also worth reading.

4.1.2 Core development mailing lists

The core Debian mailing lists that developers should use are:

- `<debian-devel-announce@lists.debian.org>`, used to announce important things to developers. All developers are expected to be subscribed to this list.
- `<debian-devel@lists.debian.org>`, used to discuss various development related technical issues.
- `<debian-policy@lists.debian.org>`, where the Debian Policy is discussed and voted on.
- `<debian-project@lists.debian.org>`, used to discuss various non-technical issues related to the project.

There are other mailing lists available for a variety of special topics; see `http://lists.debian.org/` for a list.

4.1.3 Special lists

`<debian-private@lists.debian.org>` is a special mailing list for private discussions amongst Debian developers. It is meant to be used for posts which for whatever reason should not be published publicly. As such, it is a low volume list, and users are urged not to use `<debian-private@lists.debian.org>` unless it is really necessary. Moreover, do *not* forward email from that list to anyone. Archives of this list are not available on the web for obvious reasons, but you can see them using your shell account on `lists.debian.org` and looking in the `~debian/archive/debian-private` directory.

`<debian-email@lists.debian.org>` is a special mailing list used as a grab-bag for Debian related correspondence such as contacting upstream authors about licenses, bugs, etc. or discussing the project with others where it might be useful to have the discussion archived somewhere.

4.1.4 Requesting new development-related lists

Before requesting a mailing list that relates to the development of a package (or a small group of related packages), please consider if using an alias (via a .forward-aliasname file on master.debian.org, which translates into a reasonably nice *you-aliasname@debian.org* address) or a self-managed mailing list on Alioth is more appropriate.

If you decide that a regular mailing list on lists.debian.org is really what you want, go ahead and fill in a request, following the HOWTO (`http://www.debian.org/MailingLists/HOWTO_start_list`).

4.2 IRC channels

Several IRC channels are dedicated to Debian's development. They are mainly hosted on the Open and free technology community (OFTC) (http://www.oftc.net/oftc/) network. The irc.debian.org DNS entry is an alias to irc.oftc.net.

The main channel for Debian in general is *#debian*. This is a large, general-purpose channel where users can find recent news in the topic and served by bots. *#debian* is for English speakers; there are also *#debian.de*, *#debian-fr*, *#debian-br* and other similarly named channels for speakers of other languages.

The main channel for Debian development is *#debian-devel*. It is a very active channel since usually over 150 people are always logged in. It's a channel for people who work on Debian, it's not a support channel (there's *#debian* for that). It is however open to anyone who wants to lurk (and learn). Its topic is commonly full of interesting information for developers.

Since *#debian-devel* is an open channel, you should not speak there of issues that are discussed in <debian-private@lists.debian.org>. There's another channel for this purpose, it's called *#debian-private* and it's protected by a key. This key is available in the archives of debian-private in master.debian.org:~debian/archive/debian-private/, just zgrep for *#debian-private* in all the files.

There are other additional channels dedicated to specific subjects. *#debian-bugs* is used for coordinating bug squashing parties. *#debian-boot* is used to coordinate the work on the debian-installer. *#debian-doc* is occasionally used to talk about documentation, like the document you are reading. Other channels are dedicated to an architecture or a set of packages: *#debian-bsd*, *#debian-kde*, *#debian-jr*, *#debian-edu*, *#debian-oo* (OpenOffice package) ...

Some non-English developers' channels exist as well, for example *#debian-devel-fr* for French speaking people interested in Debian's development.

Channels dedicated to Debian also exist on other IRC networks, notably on the freenode (http://www.freenode.net/) IRC network, which was pointed at by the irc.debian.org alias until 4th June 2006.

To get a cloak on freenode, you send Jörg Jaspert <joerg@debian.org> a signed mail where you tell what your nick is. Put "cloak" somewhere in the Subject: header. The nick should be registered: Nick Setup Page (http://freenode.net/faq.shtml#nicksetup). The mail needs to be signed by a key in the Debian keyring. Please see Freenodes documentation (http://freenode.net/faq.shtml#projectcloak) for more information about cloaks.

4.3 Documentation

This document contains a lot of information which is useful to Debian developers, but it cannot contain everything. Most of the other interesting documents are linked from The Developers' Corner (http://www.debian.org/devel/). Take the time to browse all the links, you will learn many more things.

4.4 Debian machines

Debian has several computers working as servers, most of which serve critical functions in the Debian project. Most of the machines are used for porting activities, and they all have a permanent connection to the Internet.

Some of the machines are available for individual developers to use, as long as the developers follow the rules set forth in the Debian Machine Usage Policies (`http://www.debian.org/devel/dmup`).

Generally speaking, you can use these machines for Debian-related purposes as you see fit. Please be kind to system administrators, and do not use up tons and tons of disk space, network bandwidth, or CPU without first getting the approval of the system administrators. Usually these machines are run by volunteers.

Please take care to protect your Debian passwords and SSH keys installed on Debian machines. Avoid login or upload methods which send passwords over the Internet in the clear, such as telnet, FTP, POP etc.

Please do not put any material that doesn't relate to Debian on the Debian servers, unless you have prior permission.

The current list of Debian machines is available at `http://db.debian.org/machines.cgi`. That web page contains machine names, contact information, information about who can log in, SSH keys etc.

If you have a problem with the operation of a Debian server, and you think that the system operators need to be notified of this problem, you can check the list of open issues in the DSA queue of our request tracker at `https://rt.debian.org/` (you can login with user "guest" and password "readonly"). To report a new problem, simply send a mail to `<admin@rt.debian.org>` and make sure to put the string "Debian RT" somewhere in the subject.

If you have a problem with a certain service, not related to the system administration (such as packages to be removed from the archive, suggestions for the web site, etc.), generally you'll report a bug against a "pseudo-package". See 'Bug reporting' on page 89 for information on how to submit bugs.

Some of the core servers are restricted, but the information from there is mirrored to another server.

4.4.1 The bugs server

`bugs.debian.org` is the canonical location for the Bug Tracking System (BTS).

It is restricted; a mirror is available on `merkel`.

If you plan on doing some statistical analysis or processing of Debian bugs, this would be the place to do it. Please describe your plans on `<debian-devel@lists.debian.org>` before implementing anything, however, to reduce unnecessary duplication of effort or wasted processing time.

4.4.2 The ftp-master server

The `ftp-master.debian.org` server holds the canonical copy of the Debian archive. Generally, package uploads go to this server; see 'Uploading a package' on page 35.

It is restricted; a mirror is available on `merkel`.

Problems with the Debian FTP archive generally need to be reported as bugs against the `ftp.debian.org` pseudo-package or an email to `<ftpmaster@debian.org>`, but also see the procedures in 'Moving, removing, renaming, adopting, and orphaning packages' on page 45.

4.4.3 The www-master server

The main web server is `www-master.debian.org`. It holds the official web pages, the face of Debian for most newbies.

If you find a problem with the Debian web server, you should generally submit a bug against the pseudo-package, `www.debian.org`. Remember to check whether or not someone else has already reported the problem to the Bug Tracking System (`http://bugs.debian.org/www.debian.org`).

4.4.4 The people web server

`people.debian.org` is the server used for developers' own web pages about anything related to Debian.

If you have some Debian-specific information which you want to serve on the web, you can do this by putting material in the `public_html` directory under your home directory on `people.debian.org`. This will be accessible at the URL `http://people.debian.org/~your-user-id/`.

You should only use this particular location because it will be backed up, whereas on other hosts it won't.

Usually the only reason to use a different host is when you need to publish materials subject to the U.S. export restrictions, in which case you can use one of the other servers located outside the United States.

Send mail to `<debian-devel@lists.debian.org>` if you have any questions.

4.4.5 The VCS servers

If you need to use a Version Control System for any of your Debian work, you can use one the existing repositories hosted on Alioth or you can request a new project and ask for the VCS repository of your choice. Alioth supports CVS (alioth.debian.org), Subversion (svn.debian.org), Arch (tla/baz, both on arch.debian.org), Bazaar (bzr.debian.org), Mercurial (hg.debian.org) and Git (git.debian.org). Checkout `http://wiki.debian.org/`

`AliothPackagingProject` if you plan to maintain packages in a VCS repository. See 'Debian's GForge installation: Alioth' on page 29 for information on the services provided by Alioth.

Historically, Debian first used `cvs.debian.org` to host CVS repositories. But that service is deprecated in favor of Alioth. Only a few projects are still using it.

4.4.6 chroots to different distributions

On some machines, there are chroots to different distributions available. You can use them like this:

```
vore% dchroot unstable
Executing shell in chroot: /org/vore.debian.org/chroots/user/unstable
```

In all chroots, the normal user home directories are available. You can find out which chroots are available via `http://db.debian.org/machines.cgi`.

4.5 The Developers Database

The Developers Database, at `https://db.debian.org/`, is an LDAP directory for managing Debian developer attributes. You can use this resource to search the list of Debian developers. Part of this information is also available through the finger service on Debian servers, try `finger yourlogin@db.debian.org` to see what it reports.

Developers can log into the database (`https://db.debian.org/login.html`) to change various information about themselves, such as:

- forwarding address for your debian.org email
- subscription to debian-private
- whether you are on vacation
- personal information such as your address, country, the latitude and longitude of the place where you live for use in the world map of Debian developers (`http://www.debian.org/devel/developers.loc`), phone and fax numbers, IRC nickname and web page
- password and preferred shell on Debian Project machines

Most of the information is not accessible to the public, naturally. For more information please read the online documentation that you can find at `http://db.debian.org/doc-general.html`.

Developers can also submit their SSH keys to be used for authorization on the official Debian machines, and even add new *.debian.net DNS entries. Those features are documented at `http://db.debian.org/doc-mail.html`.

4.6 The Debian archive

The Debian GNU/Linux distribution consists of a lot of packages (`.deb`'s, currently around 9000) and a few additional files (such as documentation and installation disk images).

Here is an example directory tree of a complete Debian archive:

```
dists/stable/main/
dists/stable/main/binary-i386/
dists/stable/main/binary-m68k/
dists/stable/main/binary-alpha/
     ...
dists/stable/main/source/
     ...
dists/stable/main/disks-i386/
dists/stable/main/disks-m68k/
dists/stable/main/disks-alpha/
     ...

dists/stable/contrib/
dists/stable/contrib/binary-i386/
dists/stable/contrib/binary-m68k/
dists/stable/contrib/binary-alpha/
     ...
dists/stable/contrib/source/

dists/stable/non-free/
dists/stable/non-free/binary-i386/
dists/stable/non-free/binary-m68k/
dists/stable/non-free/binary-alpha/
     ...
dists/stable/non-free/source/

dists/testing/
dists/testing/main/
     ...
dists/testing/contrib/
     ...
dists/testing/non-free/
     ...

dists/unstable
dists/unstable/main/
     ...
dists/unstable/contrib/
     ...
```

```
dists/unstable/non-free/
     ...

pool/
pool/main/a/
pool/main/a/apt/
     ...
pool/main/b/
pool/main/b/bash/
     ...
pool/main/liba/
pool/main/liba/libalias-perl/
     ...
pool/main/m/
pool/main/m/mailx/
     ...
pool/non-free/n/
pool/non-free/n/netscape/
     ...
```

As you can see, the top-level directory contains two directories, `dists/` and `pool/`. The latter is a "pool" in which the packages actually are, and which is handled by the archive maintenance database and the accompanying programs. The former contains the distributions, *stable*, *testing* and *unstable*. The `Packages` and `Sources` files in the distribution subdirectories can reference files in the `pool/` directory. The directory tree below each of the distributions is arranged in an identical manner. What we describe below for *stable* is equally applicable to the *unstable* and *testing* distributions.

`dists/stable` contains three directories, namely `main`, `contrib`, and `non-free`.

In each of the areas, there is a directory for the source packages (`source`) and a directory for each supported architecture (`binary-i386`, `binary-m68k`, etc.).

The `main` area contains additional directories which hold the disk images and some essential pieces of documentation required for installing the Debian distribution on a specific architecture (`disks-i386`, `disks-m68k`, etc.).

4.6.1 Sections

The *main* section of the Debian archive is what makes up the **official Debian GNU/Linux distribution**. The *main* section is official because it fully complies with all our guidelines. The other two sections do not, to different degrees; as such, they are **not** officially part of Debian GNU/Linux.

Every package in the main section must fully comply with the Debian Free Software Guidelines (`http://www.debian.org/social_contract#guidelines`) (DFSG) and with all other policy requirements as described in the Debian Policy Manual (`http://www.debian.org/`

`doc/debian-policy/`). The DFSG is our definition of "free software." Check out the Debian Policy Manual for details.

Packages in the *contrib* section have to comply with the DFSG, but may fail other requirements. For instance, they may depend on non-free packages.

Packages which do not conform to the DFSG are placed in the *non-free* section. These packages are not considered as part of the Debian distribution, though we support their use, and we provide infrastructure (such as our bug-tracking system and mailing lists) for non-free software packages.

The Debian Policy Manual (`http://www.debian.org/doc/debian-policy/`) contains a more exact definition of the three sections. The above discussion is just an introduction.

The separation of the three sections at the top-level of the archive is important for all people who want to distribute Debian, either via FTP servers on the Internet or on CD-ROMs: by distributing only the *main* and *contrib* sections, one can avoid any legal risks. Some packages in the *non-free* section do not allow commercial distribution, for example.

On the other hand, a CD-ROM vendor could easily check the individual package licenses of the packages in *non-free* and include as many on the CD-ROMs as it's allowed to. (Since this varies greatly from vendor to vendor, this job can't be done by the Debian developers.)

Note that the term "section" is also used to refer to categories which simplify the organization and browsing of available packages, e.g. *admin*, *net*, *utils* etc. Once upon a time, these sections (subsections, rather) existed in the form of subdirectories within the Debian archive. Nowadays, these exist only in the "Section" header fields of packages.

4.6.2 Architectures

In the first days, the Linux kernel was only available for Intel i386 (or greater) platforms, and so was Debian. But as Linux became more and more popular, the kernel was ported to other architectures, too.

The Linux 2.0 kernel supports Intel x86, DEC Alpha, SPARC, Motorola 680x0 (like Atari, Amiga and Macintoshes), MIPS, and PowerPC. The Linux 2.2 kernel supports even more architectures, including ARM and UltraSPARC. Since Linux supports these platforms, Debian decided that it should, too. Therefore, Debian has ports underway; in fact, we also have ports underway to non-Linux kernels. Aside from *i386* (our name for Intel x86), there is *m68k*, *alpha*, *powerpc*, *sparc*, *hurd-i386*, *arm*, *ia64*, *hppa*, *s390*, *mips*, *mipsel* and *sh* as of this writing.

Debian GNU/Linux 1.3 is only available as *i386*. Debian 2.0 shipped for *i386* and *m68k* architectures. Debian 2.1 ships for the *i386*, *m68k*, *alpha*, and *sparc* architectures. Debian 2.2 added support for the *powerpc* and *arm* architectures. Debian 3.0 added support of five new architectures: *ia64*, *hppa*, *s390*, *mips* and *mipsel*.

Information for developers and users about the specific ports are available at the Debian Ports web pages (`http://www.debian.org/ports/`).

4.6.3 Packages

There are two types of Debian packages, namely *source* and *binary* packages.

Source packages consist of either two or three files: a `.dsc` file, and either a `.tar.gz` file or both an `.orig.tar.gz` and a `.diff.gz` file.

If a package is developed specially for Debian and is not distributed outside of Debian, there is just one `.tar.gz` file which contains the sources of the program. If a package is distributed elsewhere too, the `.orig.tar.gz` file stores the so-called *upstream source code*, that is the source code that's distributed by the *upstream maintainer* (often the author of the software). In this case, the `.diff.gz` contains the changes made by the Debian maintainer.

The `.dsc` file lists all the files in the source package together with checksums (`md5sums`) and some additional info about the package (maintainer, version, etc.).

4.6.4 Distributions

The directory system described in the previous chapter is itself contained within *distribution directories*. Each distribution is actually contained in the `pool` directory in the top-level of the Debian archive itself.

To summarize, the Debian archive has a root directory within an FTP server. For instance, at the mirror site, `ftp.us.debian.org`, the Debian archive itself is contained in `/debian`, which is a common location (another is `/pub/debian`).

A distribution comprises Debian source and binary packages, and the respective `Sources` and `Packages` index files, containing the header information from all those packages. The former are kept in the `pool/` directory, while the latter are kept in the `dists/` directory of the archive (for backwards compatibility).

Stable, testing, and unstable

There are always distributions called *stable* (residing in `dists/stable`), *testing* (residing in `dists/testing`), and *unstable* (residing in `dists/unstable`). This reflects the development process of the Debian project.

Active development is done in the *unstable* distribution (that's why this distribution is sometimes called the *development distribution*). Every Debian developer can update his or her packages in this distribution at any time. Thus, the contents of this distribution change from day to day. Since no special effort is made to make sure everything in this distribution is working properly, it is sometimes literally unstable.

The "testing" distribution is generated automatically by taking packages from unstable if they satisfy certain criteria. Those criteria should ensure a good quality for packages within testing. The update to testing is launched each day after the new packages have been installed. See 'The testing distribution' on page 59.

After a period of development, once the release manager deems fit, the *testing* distribution is frozen, meaning that the policies which control how packages move from *unstable* to *testing* are tightened. Packages which are too buggy are removed. No changes are allowed into *testing* except for bug fixes. After some time has elapsed, depending on progress, the *testing* distribution is frozen even further. Details of the handling of the testing distribution are published by the Release Team on debian-devel-announce. After the open issues are solved to the satisfaction of the Release Team, the distribution is released. Releasing means that *testing* is renamed to *stable*, and a new copy is created for the new *testing*, and the previous *stable* is renamed to *oldstable* and stays there until it is finally archived. On archiving, the contents are moved to `archive.debian.org`).

This development cycle is based on the assumption that the *unstable* distribution becomes *stable* after passing a period of being in *testing*. Even once a distribution is considered stable, a few bugs inevitably remain — that's why the stable distribution is updated every now and then. However, these updates are tested very carefully and have to be introduced into the archive individually to reduce the risk of introducing new bugs. You can find proposed additions to *stable* in the `proposed-updates` directory. Those packages in `proposed-updates` that pass muster are periodically moved as a batch into the stable distribution and the revision level of the stable distribution is incremented (e.g., '3.0' becomes '3.0r1', '2.2r4' becomes '2.2r5', and so forth). Please refer to uploads to the *stable* distribution for details.

Note that development under *unstable* continues during the freeze period, since the *unstable* distribution remains in place in parallel with *testing*.

More information about the testing distribution

Packages are usually installed into the 'testing' distribution after they have undergone some degree of testing in unstable.

For more details, please see the information about the testing distribution.

Experimental

The *experimental* distribution is a special distribution. It is not a full distribution in the same sense as 'stable' and 'unstable' are. Instead, it is meant to be a temporary staging area for highly experimental software where there's a good chance that the software could break your system, or software that's just too unstable even for the *unstable* distribution (but there is a reason to package it nevertheless). Users who download and install packages from *experimental* are expected to have been duly warned. In short, all bets are off for the *experimental* distribution.

These are the `sources.list(5)` lines for *experimental*:

```
deb http://ftp.xy.debian.org/debian/ experimental main
deb-src http://ftp.xy.debian.org/debian/ experimental main
```

If there is a chance that the software could do grave damage to a system, it is likely to be better to put it into *experimental*. For instance, an experimental compressed file system should probably go into *experimental*.

Whenever there is a new upstream version of a package that introduces new features but breaks a lot of old ones, it should either not be uploaded, or be uploaded to *experimental*. A new, beta, version of some software which uses a completely different configuration can go into *experimental*, at the maintainer's discretion. If you are working on an incompatible or complex upgrade situation, you can also use *experimental* as a staging area, so that testers can get early access.

Some experimental software can still go into *unstable*, with a few warnings in the description, but that isn't recommended because packages from *unstable* are expected to propagate to *testing* and thus to *stable*. You should not be afraid to use *experimental* since it does not cause any pain to the ftpmasters, the experimental packages are automatically removed once you upload the package in *unstable* with a higher version number.

New software which isn't likely to damage your system can go directly into *unstable*.

An alternative to *experimental* is to use your personal web space on `people.debian.org`.

When uploading to unstable a package which had bugs fixed in experimental, please consider using the option `-v` to `dpkg-buildpackage` to finally get them closed.

4.6.5 Release code names

Every released Debian distribution has a *code name*: Debian 1.1 is called 'buzz'; Debian 1.2, 'rex'; Debian 1.3, 'bo'; Debian 2.0, 'hamm'; Debian 2.1, 'slink'; Debian 2.2, 'potato'; Debian 3.0, 'woody'; Debian 3.1, "sarge"; Debian 4.0, "etch". There is also a "pseudo-distribution", called 'sid', which is the current 'unstable' distribution; since packages are moved from 'unstable' to 'testing' as they approach stability, 'sid' itself is never released. As well as the usual contents of a Debian distribution, 'sid' contains packages for architectures which are not yet officially supported or released by Debian. These architectures are planned to be integrated into the mainstream distribution at some future date.

Since Debian has an open development model (i.e., everyone can participate and follow the development) even the 'unstable' and 'testing' distributions are distributed to the Internet through the Debian FTP and HTTP server network. Thus, if we had called the directory which contains the release candidate version 'testing', then we would have to rename it to 'stable' when the version is released, which would cause all FTP mirrors to re-retrieve the whole distribution (which is quite large).

On the other hand, if we called the distribution directories *Debian-x.y* from the beginning, people would think that Debian release *x.y* is available. (This happened in the past, where a CD-ROM vendor built a Debian 1.0 CD-ROM based on a pre-1.0 development version. That's the reason why the first official Debian release was 1.1, and not 1.0.)

Thus, the names of the distribution directories in the archive are determined by their code names and not their release status (e.g., 'slink'). These names stay the same during the development period and after the release; symbolic links, which can be changed easily, indicate

the currently released stable distribution. That's why the real distribution directories use the *code names*, while symbolic links for *stable*, *testing*, and *unstable* point to the appropriate release directories.

4.7 Debian mirrors

The various download archives and the web site have several mirrors available in order to relieve our canonical servers from heavy load. In fact, some of the canonical servers aren't public — a first tier of mirrors balances the load instead. That way, users always access the mirrors and get used to using them, which allows Debian to better spread its bandwidth requirements over several servers and networks, and basically makes users avoid hammering on one primary location. Note that the first tier of mirrors is as up-to-date as it can be since they update when triggered from the internal sites (we call this "push mirroring").

All the information on Debian mirrors, including a list of the available public FTP/HTTP servers, can be found at `http://www.debian.org/mirror/`. This useful page also includes information and tools which can be helpful if you are interested in setting up your own mirror, either for internal or public access.

Note that mirrors are generally run by third-parties who are interested in helping Debian. As such, developers generally do not have accounts on these machines.

4.8 The Incoming system

The Incoming system is responsible for collecting updated packages and installing them in the Debian archive. It consists of a set of directories and scripts that are installed on `ftp-master.debian.org`.

Packages are uploaded by all the maintainers into a directory called `UploadQueue`. This directory is scanned every few minutes by a daemon called `queued`, `*.command`-files are executed, and remaining and correctly signed `*.changes`-files are moved together with their corresponding files to the `unchecked` directory. This directory is not visible for most Developers, as ftp-master is restricted; it is scanned every 15 minutes by the `katie` script, which verifies the integrity of the uploaded packages and their cryptographic signatures. If the package is considered ready to be installed, it is moved into the `accepted` directory. If this is the first upload of the package (or it has new binary packages), it is moved to the `new` directory, where it waits for approval by the ftpmasters. If the package contains files to be installed "by hand" it is moved to the `byhand` directory, where it waits for manual installation by the ftpmasters. Otherwise, if any error has been detected, the package is refused and is moved to the `reject` directory.

Once the package is accepted, the system sends a confirmation mail to the maintainer and closes all the bugs marked as fixed by the upload, and the auto-builders may start recompiling it. The package is now publicly accessible at `http://incoming.debian.org/` until it is really installed in the Debian archive. This happens only once a day (and is also called the

'dinstall run' for historical reasons); the package is then removed from incoming and installed in the pool along with all the other packages. Once all the other updates (generating new `Packages` and `Sources` index files for example) have been made, a special script is called to ask all the primary mirrors to update themselves.

The archive maintenance software will also send the OpenPGP/GnuPG signed `.changes` file that you uploaded to the appropriate mailing lists. If a package is released with the `Distribution:` set to 'stable', the announcement is sent to `<debian-changes@lists.debian.org>`. If a package is released with `Distribution:` set to 'unstable' or 'experimental', the announcement will be posted to `<debian-devel-changes@lists.debian.org>` instead.

Though ftp-master is restricted, a copy of the installation is available to all developers on `merkel.debian.org`.

4.9 Package information

4.9.1 On the web

Each package has several dedicated web pages. `http://packages.debian.org/`*`package-name`* displays each version of the package available in the various distributions. Each version links to a page which provides information, including the package description, the dependencies, and package download links.

The bug tracking system tracks bugs for each package. You can view the bugs of a given package at the URL `http://bugs.debian.org/`*`package-name`*.

4.9.2 The `madison` utility

`madison` is a command-line utility that is available on `ftp-master.debian.org`, and on the mirror on `merkel.debian.org`. It uses a single argument corresponding to a package name. In result it displays which version of the package is available for each architecture and distribution combination. An example will explain it better.

```
$ madison libdbd-mysql-perl
libdbd-mysql-perl |   1.2202-4 |        stable | source, alpha, arm, i386, m6
libdbd-mysql-perl |   1.2216-2 |       testing | source, arm, hppa, i386, ia6
libdbd-mysql-perl | 1.2216-2.0.1 |       testing | alpha
libdbd-mysql-perl |   1.2219-1 |      unstable | source, alpha, arm, hppa, i3
```

In this example, you can see that the version in *unstable* differs from the version in *testing* and that there has been a binary-only NMU of the package for the alpha architecture. Each version of the package has been recompiled on most of the architectures.

4.10 The Package Tracking System

The Package Tracking System (PTS) is an email-based tool to track the activity of a source package. This really means that you can get the same emails that the package maintainer gets, simply by subscribing to the package in the PTS.

Each email sent through the PTS is classified under one of the keywords listed below. This will let you select the mails that you want to receive.

By default you will get:

`bts` All the bug reports and following discussions.

`bts-control` The email notifications from `<control@bugs.debian.org>` about bug report status changes.

`upload-source` The email notification from `katie` when an uploaded source package is accepted.

`katie-other` Other warning and error emails from `katie` (such as an override disparity for the section and/or the priority field).

`default` Any non-automatic email sent to the PTS by people who wanted to contact the subscribers of the package. This can be done by sending mail to *`sourcepackage`*`@packages.qa.debian.org`. In order to prevent spam, all messages sent to these addresses must contain the `X-PTS-Approved` header with a non-empty value.

`contact` Mails sent to the maintainer through the *@packages.debian.org email aliases.

`summary` Regular summary emails about the package's status. Currently, only progression in *testing* is sent.

You can also decide to receive additional information:

`upload-binary` The email notification from `katie` when an uploaded binary package is accepted. In other words, whenever a build daemon or a porter uploads your package for another architecture, you can get an email to track how your package gets recompiled for all architectures.

`cvs` VCS commit notifications, if the package has a VCS repository and the maintainer has set up forwarding of commit notifications to the PTS. The "cvs" name is historic, in most cases commit notifications will come from some other VCS like subversion or git.

`ddtp` Translations of descriptions or debconf templates submitted to the Debian Description Translation Project.

`derivatives` Information about changes made to the package in derivative distributions (for example Ubuntu).

4.10.1 The PTS email interface

You can control your subscription(s) to the PTS by sending various commands to <pts@qa.debian.org>.

subscribe <sourcepackage> [<email>] Subscribes *email* to communications related to the source package *sourcepackage*. Sender address is used if the second argument is not present. If *sourcepackage* is not a valid source package, you'll get a warning. However if it's a valid binary package, the PTS will subscribe you to the corresponding source package.

unsubscribe <sourcepackage> [<email>] Removes a previous subscription to the source package *sourcepackage* using the specified email address or the sender address if the second argument is left out.

unsubscribeall [<email>] Removes all subscriptions of the specified email address or the sender address if the second argument is left out.

which [<email>] Lists all subscriptions for the sender or the email address optionally specified.

keyword [<email>] Tells you the keywords that you are accepting. For an explanation of keywords, see above. Here's a quick summary:

- `bts`: mails coming from the Debian Bug Tracking System
- `bts-control`: reply to mails sent to <control@bugs.debian.org>
- `summary`: automatic summary mails about the state of a package
- `contact`: mails sent to the maintainer through the *@packages.debian.org aliases
- `cvs`: notification of VCS commits
- `ddtp`: translations of descriptions and debconf templates
- `derivatives`: changes made on the package by derivative distributions
- `upload-source`: announce of a new source upload that has been accepted
- `upload-binary`: announce of a new binary-only upload (porting)
- `katie-other`: other mails from ftpmasters (override disparity, etc.)
- `default`: all the other mails (those which aren't "automatic")

keyword <sourcepackage> [<email>] Same as the previous item but for the given source package, since you may select a different set of keywords for each source package.

keyword [<email>] {+|-|=} <list of keywords> Accept (+) or refuse (-) mails classified under the given keyword(s). Define the list (=) of accepted keywords. This changes the default set of keywords accepted by a user.

keywordall [<email>] {+|-|=} <list of keywords> Accept (+) or refuse (-) mails classified under the given keyword(s). Define the list (=) of accepted keywords. This changes the set of accepted keywords of all the currently active subscriptions of a user.

keyword <sourcepackage> [<email>] {+|-|=} <list of keywords> Same as previous item but overrides the keywords list for the indicated source package.

quit | thanks | -- Stops processing commands. All following lines are ignored by the bot.

The `pts-subscribe` command-line utility (from the `devscripts` package) can be handy to temporarily subscribe to some packages, for example after having made an non-maintainer upload.

4.10.2 Filtering PTS mails

Once you are subscribed to a package, you will get the mails sent to *sourcepackage*`@packages.qa.debian.org`. Those mails have special headers appended to let you filter them in a special mailbox (e.g. with `procmail`). The added headers are `X-Loop`, `X-PTS-Package`, `X-PTS-Keyword` and `X-Unsubscribe`.

Here is an example of added headers for a source upload notification on the `dpkg` package:

```
X-Loop: dpkg@packages.qa.debian.org
X-PTS-Package: dpkg
X-PTS-Keyword: upload-source
List-Unsubscribe: <mailto:pts@qa.debian.org?body=unsubscribe+dpkg>
```

4.10.3 Forwarding VCS commits in the PTS

If you use a publicly accessible VCS repository for maintaining your Debian package, you may want to forward the commit notification to the PTS so that the subscribers (and possible co-maintainers) can closely follow the package's evolution.

Once you set up the VCS repository to generate commit notifications, you just have to make sure it sends a copy of those mails to *sourcepackage*`_cvs@packages.qa.debian.org`. Only the people who accept the *cvs* keyword will receive these notifications. Note that the mail need to be sent from a `debian.org` machine, otherwise you'll have to add the `X-PTS-Approved: 1` header.

For Subversion repositories, the usage of svnmailer is recommended. See `http://wiki.debian.org/AliothPackagingProject` for an example on how to do it.

4.10.4 The PTS web interface

The PTS has a web interface at `http://packages.qa.debian.org/` that puts together a lot of information about each source package. It features many useful links (BTS, QA stats, contact information, DDTP translation status, buildd logs) and gathers much more information from various places (30 latest changelog entries, testing status, ...). It's a very useful tool if you

want to know what's going on with a specific source package. Furthermore there's a form that allows easy subscription to the PTS via email.

You can jump directly to the web page concerning a specific source package with a URL like `http://packages.qa.debian.org/`*`sourcepackage`*.

This web interface has been designed like a portal for the development of packages: you can add custom content on your packages' pages. You can add "static information" (news items that are meant to stay available indefinitely) and news items in the "latest news" section.

Static news items can be used to indicate:

- the availability of a project hosted on Alioth for co-maintaining the package
- a link to the upstream web site
- a link to the upstream bug tracker
- the existence of an IRC channel dedicated to the software
- any other available resource that could be useful in the maintenance of the package

Usual news items may be used to announce that:

- beta packages are available for testing
- final packages are expected for next week
- the packaging is about to be redone from scratch
- backports are available
- the maintainer is on vacation (if they wish to publish this information)
- a NMU is being worked on
- something important will affect the package

Both kinds of news are generated in a similar manner: you just have to send an email either to `<pts-static-news@qa.debian.org>` or to `<pts-news@qa.debian.org>`. The mail should indicate which package is concerned by having the name of the source package in a `X-PTS-Package` mail header or in a `Package` pseudo-header (like the BTS reports). If a URL is available in the `X-PTS-Url` mail header or in the `Url` pseudo-header, then the result is a link to that URL instead of a complete news item.

Here are a few examples of valid mails used to generate news items in the PTS. The first one adds a link to the cvsweb interface of debian-cd in the "Static information" section:

```
From: Raphael Hertzog <hertzog@debian.org>
To: pts-static-news@qa.debian.org
Subject: Browse debian-cd CVS repository with cvsweb

Package: debian-cd
Url: http://cvs.debian.org/debian-cd/
```

The second one is an announcement sent to a mailing list which is also sent to the PTS so that it is published on the PTS web page of the package. Note the use of the BCC field to avoid answers sent to the PTS by mistake.

```
From: Raphael Hertzog <hertzog@debian.org>
To: debian-gtk-gnome@lists.debian.org
Bcc: pts-news@qa.debian.org
Subject: Galeon 2.0 backported for woody
X-PTS-Package: galeon

Hello gnomers!

I'm glad to announce that galeon has been backported for woody. You'll find
everything here:
...
```

Think twice before adding a news item to the PTS because you won't be able to remove it later and you won't be able to edit it either. The only thing that you can do is send a second news item that will deprecate the information contained in the previous one.

4.11 Developer's packages overview

A QA (quality assurance) web portal is available at http://qa.debian.org/developer.php which displays a table listing all the packages of a single developer (including those where the party is listed as a co-maintainer). The table gives a good summary about the developer's packages: number of bugs by severity, list of available versions in each distribution, testing status and much more including links to any other useful information.

It is a good idea to look up your own data regularly so that you don't forget any open bugs, and so that you don't forget which packages are your responsibility.

4.12 Debian's GForge installation: Alioth

Alioth is a Debian service based on a slightly modified version of the GForge software (which evolved from SourceForge). This software offers developers access to easy-to-use tools such as

bug trackers, patch manager, project/task managers, file hosting services, mailing lists, CVS repositories etc. All these tools are managed via a web interface.

It is intended to provide facilities to free software projects backed or led by Debian, facilitate contributions from external developers to projects started by Debian, and help projects whose goals are the promotion of Debian or its derivatives. It's heavily used by many Debian teams and provides hosting for all sorts of VCS repositories.

All Debian developers automatically have an account on Alioth. They can activate it by using the recover password facility. External developers can request guest accounts on Alioth.

For more information please visit the following links:

- http://wiki.debian.org/Alioth
- http://wiki.debian.org/AliothFAQ
- http://wiki.debian.org/AliothPackagingProject
- http://alioth.debian.org/

4.13 Goodies for Developers

4.13.1 LWN Subscriptions

Since October of 2002, HP has sponsored a subscription to LWN for all interested Debian developers. Details on how to get access to this benefit are in http://lists.debian.org/debian-devel-announce/2002/10/msg00018.html.

Chapter 5

Managing Packages

This chapter contains information related to creating, uploading, maintaining, and porting packages.

5.1 New packages

If you want to create a new package for the Debian distribution, you should first check the Work-Needing and Prospective Packages (WNPP) (`http://www.debian.org/devel/wnpp/`) list. Checking the WNPP list ensures that no one is already working on packaging that software, and that effort is not duplicated. Read the WNPP web pages (`http://www.debian.org/devel/wnpp/`) for more information.

Assuming no one else is already working on your prospective package, you must then submit a bug report ('Bug reporting' on page 89) against the pseudo-package `wnpp` describing your plan to create a new package, including, but not limiting yourself to, a description of the package, the license of the prospective package, and the current URL where it can be downloaded from.

You should set the subject of the bug to "ITP: *foo* – *short description*", substituting the name of the new package for *foo*. The severity of the bug report must be set to *wishlist*. If you feel it's necessary, send a copy to `<debian-devel@lists.debian.org>` by putting the address in the `X-Debbugs-CC:` header of the message (no, don't use `CC:`, because that way the message's subject won't indicate the bug number).

Please include a `Closes: bug#`*nnnnn* entry in the changelog of the new package in order for the bug report to be automatically closed once the new package is installed in the archive (see 'When bugs are closed by new uploads' on page 40).

When closing security bugs include CVE numbers as well as the "Closes: #nnnnn". This is useful for the security team to track vulnerabilities. If an upload is made to fix the bug before the advisory ID is known, it is encouraged to modify the historical changelog entry with the next upload. Even in this case, please include all available pointers to background information in the original changelog entry.

There are a number of reasons why we ask maintainers to announce their intentions:

- It helps the (potentially new) maintainer to tap into the experience of people on the list, and lets them know if anyone else is working on it already.
- It lets other people thinking about working on the package know that there already is a volunteer, so efforts may be shared.
- It lets the rest of the maintainers know more about the package than the one line description and the usual changelog entry "Initial release" that gets posted to `debian-devel-changes`.
- It is helpful to the people who live off unstable (and form our first line of testers). We should encourage these people.
- The announcements give maintainers and other interested parties a better feel of what is going on, and what is new, in the project.

Please see `http://ftp-master.debian.org/REJECT-FAQ.html` for common rejection reasons for a new package.

5.2 Recording changes in the package

Changes that you make to the package need to be recorded in the `debian/changelog`. These changes should provide a concise description of what was changed, why (if it's in doubt), and note if any bugs were closed. They also record when the package was completed. This file will be installed in `/usr/share/doc/`*`package`*`/changelog.Debian.gz`, or `/usr/share/doc/`*`package`*`/changelog.gz` for native packages.

The `debian/changelog` file conforms to a certain structure, with a number of different fields. One field of note, the *distribution*, is described in 'Picking a distribution' on page 34. More information about the structure of this file can be found in the Debian Policy section titled "`debian/changelog`".

Changelog entries can be used to automatically close Debian bugs when the package is installed into the archive. See 'When bugs are closed by new uploads' on page 40.

It is conventional that the changelog entry of a package that contains a new upstream version of the software looks like this:

```
* new upstream version
```

There are tools to help you create entries and finalize the `changelog` for release — see '`devscripts`' on page 104 and '`dpkg-dev-el`' on page 105.

See also 'Best practices for `debian/changelog`' on page 71.

5.3 Testing the package

Before you upload your package, you should do basic testing on it. At a minimum, you should try the following activities (you'll need to have an older version of the same Debian package around):

- Install the package and make sure the software works, or upgrade the package from an older version to your new version if a Debian package for it already exists.
- Run `lintian` over the package. You can run `lintian` as follows: `lintian -v` *`package-version`*`.changes`. This will check the source package as well as the binary package. If you don't understand the output that `lintian` generates, try adding the `-i` switch, which will cause `lintian` to output a very verbose description of the problem.

  Normally, a package should *not* be uploaded if it causes lintian to emit errors (they will start with `E`).

  For more information on `lintian`, see '`lintian`' on page 100.
- Optionally run '`debdiff`' on page 101 to analyze changes from an older version, if one exists.
- Downgrade the package to the previous version (if one exists) — this tests the `postrm` and `prerm` scripts.
- Remove the package, then reinstall it.
- Copy the source package in a different directory and try unpacking it and rebuilding it. This tests if the package relies on existing files outside of it, or if it relies on permissions being preserved on the files shipped inside the .diff.gz file.

5.4 Layout of the source package

There are two types of Debian source packages:

- the so-called *native* packages, where there is no distinction between the original sources and the patches applied for Debian
- the (more common) packages where there's an original source tarball file accompanied by another file that contains the patches applied for Debian

For the native packages, the source package includes a Debian source control file (`.dsc`) and the source tarball (`.tar.gz`). A source package of a non-native package includes a Debian source control file, the original source tarball (`.orig.tar.gz`) and the Debian patches (`.diff.gz`).

Whether a package is native or not is determined when it is built by `dpkg-buildpackage(1)`. The rest of this section relates only to non-native packages.

The first time a version is uploaded which corresponds to a particular upstream version, the original source tar file should be uploaded and included in the `.changes` file. Subsequently, this very same tar file should be used to build the new diffs and `.dsc` files, and will not need to be re-uploaded.

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

If no original source is included in the upload, the original source tar-file used by `dpkg-source` when constructing the `.dsc` file and diff to be uploaded *must* be byte-for-byte identical with the one already in the archive.

Please notice that, in non-native packages, permissions on files that are not present in the .orig.tar.gz will not be preserved, as diff does not store file permissions in the patch.

5.5 Picking a distribution

Each upload needs to specify which distribution the package is intended for. The package build process extracts this information from the first line of the `debian/changelog` file and places it in the `Distribution` field of the `.changes` file.

There are several possible values for this field: 'stable', 'unstable', 'testing-proposed-updates' and 'experimental'. Normally, packages are uploaded into *unstable*.

Actually, there are two other possible distributions: 'stable-security' and 'testing-security', but read 'Handling security-related bugs' on page 41 for more information on those.

It is not possible to upload a package into several distributions at the same time.

5.5.1 Special case: uploads to the *stable* distribution

Uploading to *stable* means that the package will transfered to the *p-u-new*-queue for review by the stable release managers, and if approved will be installed in `stable-proposed-updates` directory of the Debian archive. From there, it will be included in *stable* with the next point release.

Extra care should be taken when uploading to *stable*. Basically, a package should only be uploaded to stable if one of the following happens:

- a truly critical functionality problem
- the package becomes uninstallable
- a released architecture lacks the package

In the past, uploads to *stable* were used to address security problems as well. However, this practice is deprecated, as uploads used for Debian security advisories are automatically copied to the appropriate `proposed-updates` archive when the advisory is released. See 'Handling security-related bugs' on page 41 for detailed information on handling security problems.

Changing anything else in the package that isn't important is discouraged, because even trivial fixes can cause bugs later on.

Packages uploaded to *stable* need to be compiled on systems running *stable*, so that their dependencies are limited to the libraries (and other packages) available in *stable*; for example, a package uploaded to *stable* that depends on a library package that only exists in unstable will be rejected. Making changes to dependencies of other packages (by messing with `Provides` or shlibs files), possibly making those other packages uninstallable, is strongly discouraged.

The Release Team (which can be reached at `<debian-release@lists.debian.org>`) will regularly evaluate the uploads To *stable-proposed-updates* and decide if your package can be included in *stable*. Please be clear (and verbose, if necessary) in your changelog entries for uploads to *stable*, because otherwise the package won't be considered for inclusion.

It's best practice to speak with the stable release manager *before* uploading to *stable*/*stable-proposed-updates*, so that the uploaded package fits the needs of the next point release.

5.5.2 Special case: uploads to *testing/testing-proposed-updates*

Please see the information in the testing section for details.

5.6 Uploading a package

5.6.1 Uploading to `ftp-master`

To upload a package, you should upload the files (including the signed changes and dsc-file) with anonymous ftp to `ftp-master.debian.org` in the directory `/pub/UploadQueue/`. To get the files processed there, they need to be signed with a key in the debian keyring.

Please note that you should transfer the changes file last. Otherwise, your upload may be rejected because the archive maintenance software will parse the changes file and see that not all files have been uploaded.

You may also find the Debian packages '`dupload`' on page 103 or '`dput`' on page 103 useful when uploading packages. These handy programs help automate the process of uploading packages into Debian.

For removing packages, please see the README file in that ftp directory, and the Debian package '`dcut`' on page 104.

5.6.2 Delayed uploads

Delayed uploads are done for the moment via the delayed queue at gluck. The upload-directory is `gluck:~tfheen/DELAYED/[012345678]-day`. 0-day is uploaded multiple times per day to ftp-master.

With a fairly recent dput, this section

```
[tfheen_delayed]
method = scp
fqdn = gluck.debian.org
incoming = ~tfheen
```

in ~/.dput.cf should work fine for uploading to the DELAYED queue.

Note: Since this upload queue goes to `ftp-master`, the prescription found in 'Uploading to `ftp-master`' on the page before applies here as well.

5.6.3 Security uploads

Do **NOT** upload a package to the security upload queue (oldstable-security, stable-security, etc.) without prior authorization from the security team. If the package does not exactly meet the team's requirements, it will cause many problems and delays in dealing with the unwanted upload. For details, please see section 'Handling security-related bugs' on page 41.

5.6.4 Other upload queues

The scp queues on ftp-master, and security are mostly unusable due to the login restrictions on those hosts.

The anonymous queues on ftp.uni-erlangen.de and ftp.uk.debian.org are currently down. Work is underway to resurrect them.

The queues on master.debian.org, samosa.debian.org, master.debian.or.jp, and ftp.chiark.greenend.org.uk are down permanently, and will not be resurrected. The queue in Japan will be replaced with a new queue on hp.debian.or.jp some day.

For the time being, the anonymous ftp queue on auric.debian.org (the former ftp-master) works, but it is deprecated and will be removed at some point in the future.

5.6.5 Notification that a new package has been installed

The Debian archive maintainers are responsible for handling package uploads. For the most part, uploads are automatically handled on a daily basis by the archive maintenance tools, `katie`. Specifically, updates to existing packages to the 'unstable' distribution are handled automatically. In other cases, notably new packages, placing the uploaded package into the distribution is handled manually. When uploads are handled manually, the change to the archive may take up to a month to occur. Please be patient.

In any case, you will receive an email notification indicating that the package has been added to the archive, which also indicates which bugs will be closed by the upload. Please examine this notification carefully, checking if any bugs you meant to close didn't get triggered.

The installation notification also includes information on what section the package was inserted into. If there is a disparity, you will receive a separate email notifying you of that. Read on below.

Note that if you upload via queues, the queue daemon software will also send you a notification by email.

5.7 Specifying the package section, subsection and priority

The `debian/control` file's `Section` and `Priority` fields do not actually specify where the file will be placed in the archive, nor its priority. In order to retain the overall integrity of the archive, it is the archive maintainers who have control over these fields. The values in the `debian/control` file are actually just hints.

The archive maintainers keep track of the canonical sections and priorities for packages in the *override file*. If there is a disparity between the *override file* and the package's fields as indicated in `debian/control`, then you will receive an email noting the divergence when the package is installed into the archive. You can either correct your `debian/control` file for your next upload, or else you may wish to make a change in the *override file*.

To alter the actual section that a package is put in, you need to first make sure that the `debian/control` file in your package is accurate. Next, send an email `<override-change@debian.org>` or submit a bug against `ftp.debian.org` requesting that the section or priority for your package be changed from the old section or priority to the new one. Be sure to explain your reasoning.

For more information about *override files*, see `dpkg-scanpackages(1)` and `http://www.debian.org/Bugs/Developer#maintincorrect`.

Note that the `Section` field describes both the section as well as the subsection, which are described in 'Sections' on page 18. If the section is "main", it should be omitted. The list of allowable subsections can be found in `http://www.debian.org/doc/debian-policy/ch-archive.html#s-subsections`.

5.8 Handling bugs

Every developer has to be able to work with the Debian bug tracking system (`http://www.debian.org/Bugs/`). This includes knowing how to file bug reports properly (see 'Bug reporting' on page 89), how to update them and reorder them, and how to process and close them.

The bug tracking system's features are described in the BTS documentation for developers (`http://www.debian.org/Bugs/Developer`). This includes closing bugs, sending followup messages, assigning severities and tags, marking bugs as forwarded, and other issues.

Operations such as reassigning bugs to other packages, merging separate bug reports about the same issue, or reopening bugs when they are prematurely closed, are handled using the

so-called control mail server. All of the commands available on this server are described in the BTS control server documentation (http://www.debian.org/Bugs/server-control).

5.8.1 Monitoring bugs

If you want to be a good maintainer, you should periodically check the Debian bug tracking system (BTS) (http://www.debian.org/Bugs/) for your packages. The BTS contains all the open bugs against your packages. You can check them by browsing this page: http://bugs.debian.org/*yourlogin*@debian.org.

Maintainers interact with the BTS via email addresses at bugs.debian.org. Documentation on available commands can be found at http://www.debian.org/Bugs/, or, if you have installed the doc-debian package, you can look at the local files /usr/share/doc/debian/bug-*.

Some find it useful to get periodic reports on open bugs. You can add a cron job such as the following if you want to get a weekly email outlining all the open bugs against your packages:

```
# ask for weekly reports of bugs in my packages
0 17 * * fri   echo "index maint address" | mail request@bugs.debian.org
```

Replace *address* with your official Debian maintainer address.

5.8.2 Responding to bugs

When responding to bugs, make sure that any discussion you have about bugs is sent both to the original submitter of the bug, and to the bug itself (e.g., <123@bugs.debian.org>). If you're writing a new mail and you don't remember the submitter email address, you can use the <123-submitter@bugs.debian.org> email to contact the submitter *and* to record your mail within the bug log (that means you don't need to send a copy of the mail to <123@bugs.debian.org>).

If you get a bug which mentions "FTBFS", this means "Fails to build from source". Porters frequently use this acronym.

Once you've dealt with a bug report (e.g. fixed it), mark it as *done* (close it) by sending an explanation message to <123-done@bugs.debian.org>. If you're fixing a bug by changing and uploading the package, you can automate bug closing as described in 'When bugs are closed by new uploads' on page 40.

You should *never* close bugs via the bug server close command sent to <control@bugs.debian.org>. If you do so, the original submitter will not receive any information about why the bug was closed.

5.8.3 Bug housekeeping

As a package maintainer, you will often find bugs in other packages or have bugs reported against your packages which are actually bugs in other packages. The bug tracking system's features are described in the BTS documentation for Debian developers (`http://www.debian.org/Bugs/Developer`). Operations such as reassigning, merging, and tagging bug reports are described in the BTS control server documentation (`http://www.debian.org/Bugs/server-control`). This section contains some guidelines for managing your own bugs, based on the collective Debian developer experience.

Filing bugs for problems that you find in other packages is one of the "civic obligations" of maintainership, see 'Bug reporting' on page 89 for details. However, handling the bugs in your own packages is even more important.

Here's a list of steps that you may follow to handle a bug report:

1 Decide whether the report corresponds to a real bug or not. Sometimes users are just calling a program in the wrong way because they haven't read the documentation. If you diagnose this, just close the bug with enough information to let the user correct their problem (give pointers to the good documentation and so on). If the same report comes up again and again you may ask yourself if the documentation is good enough or if the program shouldn't detect its misuse in order to give an informative error message. This is an issue that may need to be brought up with the upstream author.

   If the bug submitter disagrees with your decision to close the bug, they may reopen it until you find an agreement on how to handle it. If you don't find any, you may want to tag the bug `wontfix` to let people know that the bug exists but that it won't be corrected. If this situation is unacceptable, you (or the submitter) may want to require a decision of the technical committee by reassigning the bug to `tech-ctte` (you may use the clone command of the BTS if you wish to keep it reported against your package). Before doing so, please read the recommended procedure (`http://www.debian.org/devel/tech-ctte`).

2 If the bug is real but it's caused by another package, just reassign the bug to the right package. If you don't know which package it should be reassigned to, you should ask for help on IRC or on `<debian-devel@lists.debian.org>`. Please make sure that the maintainer(s) of the package the bug is reassigned to know why you reassigned it.

   Sometimes you also have to adjust the severity of the bug so that it matches our definition of the severity. That's because people tend to inflate the severity of bugs to make sure their bugs are fixed quickly. Some bugs may even be dropped to wishlist severity when the requested change is just cosmetic.

3 If the bug is real but the same problem has already been reported by someone else, then the two relevant bug reports should be merged into one using the merge command of the BTS. In this way, when the bug is fixed, all of the submitters will be informed of this. (Note, however, that emails sent to one bug report's submitter won't automatically be sent to the other report's submitter.) For more details on the technicalities of the merge

command and its relative, the unmerge command, see the BTS control server documentation.

4 The bug submitter may have forgotten to provide some information, in which case you have to ask them for the required information. You may use the `moreinfo` tag to mark the bug as such. Moreover if you can't reproduce the bug, you tag it `unreproducible`. Anyone who can reproduce the bug is then invited to provide more information on how to reproduce it. After a few months, if this information has not been sent by someone, the bug may be closed.

5 If the bug is related to the packaging, you just fix it. If you are not able to fix it yourself, then tag the bug as `help`. You can also ask for help on `<debian-devel@lists.debian.org>` or `<debian-qa@lists.debian.org>`. If it's an upstream problem, you have to forward it to the upstream author. Forwarding a bug is not enough, you have to check at each release if the bug has been fixed or not. If it has, you just close it, otherwise you have to remind the author about it. If you have the required skills you can prepare a patch that fixes the bug and send it to the author at the same time. Make sure to send the patch to the BTS and to tag the bug as `patch`.

6 If you have fixed a bug in your local copy, or if a fix has been committed to the CVS repository, you may tag the bug as `pending` to let people know that the bug is corrected and that it will be closed with the next upload (add the `closes:` in the `changelog`). This is particularly useful if you are several developers working on the same package.

7 Once a corrected package is available in the *unstable* distribution, you can close the bug. This can be done automatically, read 'When bugs are closed by new uploads' on the current page.

5.8.4 When bugs are closed by new uploads

As bugs and problems are fixed in your packages, it is your responsibility as the package maintainer to close these bugs. However, you should not close a bug until the package which fixes the bug has been accepted into the Debian archive. Therefore, once you get notification that your updated package has been installed into the archive, you can and should close the bug in the BTS. Also, the bug should be closed with the correct version.

However, it's possible to avoid having to manually close bugs after the upload — just list the fixed bugs in your `debian/changelog` file, following a certain syntax, and the archive maintenance software will close the bugs for you. For example:

```
acme-cannon (3.1415) unstable; urgency=low

  * Frobbed with options (closes: Bug#98339)
  * Added safety to prevent operator dismemberment, closes: bug#98765,
    bug#98713, #98714.
  * Added man page. Closes: #98725.
```

Technically speaking, the following Perl regular expression describes how bug closing changelogs are identified:

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

We prefer the `closes: #XXX` syntax, as it is the most concise entry and the easiest to integrate with the text of the `changelog`. Unless specified different by the *-v*-switch to `dpkg-buildpackage`, only the bugs closed in the most recent changelog entry are closed (basically, exactly the bugs mentioned in the changelog-part in the `.changes` file are closed).

Historically, uploads identified as Non-maintainer upload (NMU) were tagged `fixed` instead of being closed, but that practice was ceased with the advent of version-tracking. The same applied to the tag `fixed-in-experimental`.

If you happen to mistype a bug number or forget a bug in the changelog entries, don't hesitate to undo any damage the error caused. To reopen wrongly closed bugs, send a `reopen XXX` command to the bug tracking system's control address, `<control@bugs.debian.org>`. To close any remaining bugs that were fixed by your upload, email the `.changes` file to `<XXX-done@bugs.debian.org>`, where *XXX* is the bug number, and put "Version: YYY" and an empty line as the first two lines of the body of the email, where *YYY* is the first version where the bug has been fixed.

Bear in mind that it is not obligatory to close bugs using the changelog as described above. If you simply want to close bugs that don't have anything to do with an upload you made, do it by emailing an explanation to `<XXX-done@bugs.debian.org>`. Do **not** close bugs in the changelog entry of a version if the changes in that version of the package don't have any bearing on the bug.

For general information on how to write your changelog entries, see 'Best practices for `debian/changelog`' on page 71.

5.8.5 Handling security-related bugs

Due to their sensitive nature, security-related bugs must be handled carefully. The Debian Security Team exists to coordinate this activity, keeping track of outstanding security problems, helping maintainers with security problems or fixing them themselves, sending security advisories, and maintaining security.debian.org.

When you become aware of a security-related bug in a Debian package, whether or not you are the maintainer, collect pertinent information about the problem, and promptly contact the security team at `<team@security.debian.org>` as soon as possible. **DO NOT UPLOAD** any packages for stable; the security team will do that. Useful information includes, for example:

- Which versions of the package are known to be affected by the bug. Check each version that is present in a supported Debian release, as well as testing and unstable.
- The nature of the fix, if any is available (patches are especially helpful)

- Any fixed packages that you have prepared yourself (send only the `.diff.gz` and `.dsc` files and read 'Preparing packages to address security issues' on the next page first)
- Any assistance you can provide to help with testing (exploits, regression testing, etc.)
- Any information needed for the advisory (see 'Security Advisories' on this page)

Confidentiality

Unlike most other activities within Debian, information about security issues must sometimes be kept private for a time. This allows software distributors to coordinate their disclosure in order to minimize their users' exposure. Whether this is the case depends on the nature of the problem and corresponding fix, and whether it is already a matter of public knowledge.

There are several ways developers can learn of a security problem:

- they notice it on a public forum (mailing list, web site, etc.)
- someone files a bug report
- someone informs them via private email

In the first two cases, the information is public and it is important to have a fix as soon as possible. In the last case, however, it might not be public information. In that case there are a few possible options for dealing with the problem:

- If the security exposure is minor, there is sometimes no need to keep the problem a secret and a fix should be made and released.
- If the problem is severe, it is preferable to share the information with other vendors and coordinate a release. The security team keeps in contact with the various organizations and individuals and can take care of that.

In all cases if the person who reports the problem asks that it not be disclosed, such requests should be honored, with the obvious exception of informing the security team in order that a fix may be produced for a stable release of Debian. When sending confidential information to the security team, be sure to mention this fact.

Please note that if secrecy is needed you may not upload a fix to unstable (or anywhere else, such as a public CVS repository). It is not sufficient to obfuscate the details of the change, as the code itself is public, and can (and will) be examined by the general public.

There are two reasons for releasing information even though secrecy is requested: the problem has been known for a while, or the problem or exploit has become public.

Security Advisories

Security advisories are only issued for the current, released stable distribution, and *not* for testing or unstable. When released, advisories are sent to the `<debian-security-announce@lists.debian.org>` mailing list and posted on the security web page (`http://www.debian.org/security/`). Security advisories are written and posted by the security team. However they certainly do not mind if a maintainer can supply some of the information for them, or write part of the text. Information that should be in an advisory includes:

- A description of the problem and its scope, including:
  - The type of problem (privilege escalation, denial of service, etc.)
  - What privileges may be gained, and by whom (if any)
  - How it can be exploited
  - Whether it is remotely or locally exploitable
  - How the problem was fixed

  This information allows users to assess the threat to their systems.
- Version numbers of affected packages
- Version numbers of fixed packages
- Information on where to obtain the updated packages (usually from the Debian security archive)
- References to upstream advisories, CVE (`http://cve.mitre.org`) identifiers, and any other information useful in cross-referencing the vulnerability

Preparing packages to address security issues

One way that you can assist the security team in their duties is to provide them with fixed packages suitable for a security advisory for the stable Debian release.

When an update is made to the stable release, care must be taken to avoid changing system behavior or introducing new bugs. In order to do this, make as few changes as possible to fix the bug. Users and administrators rely on the exact behavior of a release once it is made, so any change that is made might break someone's system. This is especially true of libraries: make sure you never change the API or ABI, no matter how small the change.

This means that moving to a new upstream version is not a good solution. Instead, the relevant changes should be back-ported to the version present in the current stable Debian release. Generally, upstream maintainers are willing to help if needed. If not, the Debian security team may be able to help.

In some cases, it is not possible to back-port a security fix, for example when large amounts of source code need to be modified or rewritten. If this happens, it may be necessary to move to a new upstream version. However, this is only done in extreme situations, and you must always coordinate that with the security team beforehand.

Related to this is another important guideline: always test your changes. If you have an exploit available, try it and see if it indeed succeeds on the unpatched package and fails on the fixed package. Test other, normal actions as well, as sometimes a security fix can break seemingly unrelated features in subtle ways.

Do **NOT** include any changes in your package which are not directly related to fixing the vulnerability. These will only need to be reverted, and this wastes time. If there are other bugs in your package that you would like to fix, make an upload to proposed-updates in the usual way, after the security advisory is issued. The security update mechanism is not a means for introducing changes to your package which would otherwise be rejected from the stable release, so please do not attempt to do this.

Review and test your changes as much as possible. Check the differences from the previous version repeatedly (`interdiff` from the `patchutils` package and `debdiff` from

`devscripts` are useful tools for this, see '`debdiff`' on page 101).

Be sure to verify the following items:

- Target the right distribution in your `debian/changelog`. For stable this is `stable-security` and for testing this is `testing-security`, and for the previous stable release, this is `oldstable-security`. Do not target *distribution*-proposed-updates or `stable`!
- The upload should have urgency=high.
- Make descriptive, meaningful changelog entries. Others will rely on them to determine whether a particular bug was fixed. Always include an external reference, preferably a CVE identifier, so that it can be cross-referenced. Include the same information in the changelog for unstable, so that it is clear that the same bug was fixed, as this is very helpful when verifying that the bug is fixed in the next stable release. If a CVE identifier has not yet been assigned, the security team will request one so that it can be included in the package and in the advisory.
- Make sure the version number is proper. It must be greater than the current package, but less than package versions in later distributions. If in doubt, test it with `dpkg --compare-versions`. Be careful not to re-use a version number that you have already used for a previous upload. For *testing*, there must be a higher version in *unstable*. If there is none yet (for example, if *testing* and *unstable* have the same version) you must upload a new version to unstable first.
- Do not make source-only uploads if your package has any binary-all packages (do not use the `-S` option to `dpkg-buildpackage`). The `buildd` infrastructure will not build those. This point applies to normal package uploads as well.
- Unless the upstream source has been uploaded to security.debian.org before (by a previous security update), build the upload with full upstream source (`dpkg-buildpackage -sa`). If there has been a previous upload to security.debian.org with the same upstream version, you may upload without upstream source (`dpkg-buildpackage -sd`).
- Be sure to use the exact same `*.orig.tar.gz` as used in the normal archive, otherwise it is not possible to move the security fix into the main archives later.
- Build the package on a clean system which only has packages installed from the distribution you are building for. If you do not have such a system yourself, you can use a debian.org machine (see 'Debian machines' on page 14) or setup a chroot (see '`pbuilder`' on page 103 and '`debootstrap`' on page 103).

Uploading the fixed package

Do **NOT** upload a package to the security upload queue (oldstable-security, stable-security, etc.) without prior authorization from the security team. If the package does not exactly meet

the team's requirements, it will cause many problems and delays in dealing with the unwanted upload.

Do **NOT** upload your fix to proposed-updates without coordinating with the security team. Packages from security.debian.org will be copied into the proposed-updates directory automatically. If a package with the same or a higher version number is already installed into the archive, the security update will be rejected by the archive system. That way, the stable distribution will end up without a security update for this package instead.

Once you have created and tested the new package and it has been approved by the security team, it needs to be uploaded so that it can be installed in the archives. For security uploads, the place to upload to is `ftp://security-master.debian.org/pub/SecurityUploadQueue/` .

Once an upload to the security queue has been accepted, the package will automatically be rebuilt for all architectures and stored for verification by the security team.

Uploads which are waiting for acceptance or verification are only accessible by the security team. This is necessary since there might be fixes for security problems that cannot be disclosed yet.

If a member of the security team accepts a package, it will be installed on security.debian.org as well as proposed for the proper *distribution*-proposed-updates on ftp-master.

5.9 Moving, removing, renaming, adopting, and orphaning packages

Some archive manipulation operations are not automated in the Debian upload process. These procedures should be manually followed by maintainers. This chapter gives guidelines on what to do in these cases.

5.9.1 Moving packages

Sometimes a package will change its section. For instance, a package from the 'non-free' section might be GPL'd in a later version, in which case the package should be moved to 'main' or 'contrib'.[1]

If you need to change the section for one of your packages, change the package control information to place the package in the desired section, and re-upload the package (see the Debian Policy Manual (`http://www.debian.org/doc/debian-policy/`) for details). You must ensure that you include the `.orig.tar.gz` in your upload (even if you are not uploading a new upstream version), or it will not appear in the new section together with the rest of the package. If your new section is valid, it will be moved automatically. If it does not, then contact the ftpmasters in order to understand what happened.

[1] See the Debian Policy Manual (`http://www.debian.org/doc/debian-policy/`) for guidelines on what section a package belongs in.

If, on the other hand, you need to change the *subsection* of one of your packages (e.g., "devel", "admin"), the procedure is slightly different. Correct the subsection as found in the control file of the package, and re-upload that. Also, you'll need to get the override file updated, as described in 'Specifying the package section, subsection and priority' on page 37.

5.9.2 Removing packages

If for some reason you want to completely remove a package (say, if it is an old compatibility library which is no longer required), you need to file a bug against `ftp.debian.org` asking that the package be removed; as all bugs, this bug should normally have normal severity. Make sure you indicate which distribution the package should be removed from. Normally, you can only have packages removed from *unstable* and *experimental*. Packages are not removed from *testing* directly. Rather, they will be removed automatically after the package has been removed from *unstable* and no package in *testing* depends on it.

There is one exception when an explicit removal request is not necessary: If a (source or binary) package is an orphan, it will be removed semi-automatically. For a binary-package, this means if there is no longer any source package producing this binary package; if the binary package is just no longer produced on some architectures, a removal request is still necessary. For a source-package, this means that all binary packages it refers to have been taken over by another source package.

In your removal request, you have to detail the reasons justifying the request. This is to avoid unwanted removals and to keep a trace of why a package has been removed. For example, you can provide the name of the package that supersedes the one to be removed.

Usually you only ask for the removal of a package maintained by yourself. If you want to remove another package, you have to get the approval of its maintainer.

Further information relating to these and other package removal related topics may be found at `http://wiki.debian.org/ftpmaster_Removals` and `http://qa.debian.org/howto-remove.html`.

If in doubt concerning whether a package is disposable, email `<debian-devel@lists.debian.org>` asking for opinions. Also of interest is the `apt-cache` program from the `apt` package. When invoked as `apt-cache showpkg` *`package`*, the program will show details for *package*, including reverse depends. Other useful programs include `apt-cache rdepends`, `apt-rdepends` and `grep-dctrl`. Removal of orphaned packages is discussed on `<debian-qa@lists.debian.org>`.

Once the package has been removed, the package's bugs should be handled. They should either be reassigned to another package in the case where the actual code has evolved into another package (e.g. `libfoo12` was removed because `libfoo13` supersedes it) or closed if the software is simply no longer part of Debian.

Removing packages from `Incoming`

In the past, it was possible to remove packages from `incoming`. However, with the introduction of the new incoming system, this is no longer possible. Instead, you have to upload a new revision of your package with a higher version than the package you want to replace. Both versions will be installed in the archive but only the higher version will actually be available in *unstable* since the previous version will immediately be replaced by the higher. However, if you do proper testing of your packages, the need to replace a package should not occur too often anyway.

5.9.3 Replacing or renaming packages

When you make a mistake naming your package, you should follow a two-step process to rename it. First, set your `debian/control` file to replace and conflict with the obsolete name of the package (see the Debian Policy Manual (`http://www.debian.org/doc/debian-policy/`) for details). Once you've uploaded the package and the package has moved into the archive, file a bug against `ftp.debian.org` asking to remove the package with the obsolete name. Do not forget to properly reassign the package's bugs at the same time.

At other times, you may make a mistake in constructing your package and wish to replace it. The only way to do this is to increase the version number and upload a new version. The old version will be expired in the usual manner. Note that this applies to each part of your package, including the sources: if you wish to replace the upstream source tarball of your package, you will need to upload it with a different version. An easy possibility is to replace `foo_1.00.orig.tar.gz` with `foo_1.00+0.orig.tar.gz`. This restriction gives each file on the ftp site a unique name, which helps to ensure consistency across the mirror network.

5.9.4 Orphaning a package

If you can no longer maintain a package, you need to inform others, and see that the package is marked as orphaned. You should set the package maintainer to `Debian QA Group <packages@qa.debian.org>` and submit a bug report against the pseudo package `wnpp`. The bug report should be titled `O: package -- short description` indicating that the package is now orphaned. The severity of the bug should be set to *normal*; if the package has a priority of standard or higher, it should be set to important. If you feel it's necessary, send a copy to `<debian-devel@lists.debian.org>` by putting the address in the X-Debbugs-CC: header of the message (no, don't use CC:, because that way the message's subject won't indicate the bug number).

If you just intend to give the package away, but you can keep maintainership for the moment, then you should instead submit a bug against `wnpp` and title it `RFA: package -- short description`. `RFA` stands for *Request For Adoption*.

More information is on the WNPP web pages (`http://www.debian.org/devel/wnpp/`).

5.9.5 Adopting a package

A list of packages in need of a new maintainer is available in the Work-Needing and Prospective Packages list (WNPP) (`http://www.debian.org/devel/wnpp/`). If you wish to take over maintenance of any of the packages listed in the WNPP, please take a look at the aforementioned page for information and procedures.

It is not OK to simply take over a package that you feel is neglected — that would be package hijacking. You can, of course, contact the current maintainer and ask them if you may take over the package. If you have reason to believe a maintainer has gone AWOL (absent without leave), see 'Dealing with inactive and/or unreachable maintainers' on page 91.

Generally, you may not take over the package without the assent of the current maintainer. Even if they ignore you, that is still not grounds to take over a package. Complaints about maintainers should be brought up on the developers' mailing list. If the discussion doesn't end with a positive conclusion, and the issue is of a technical nature, consider bringing it to the attention of the technical committee (see the technical committee web page (`http://www.debian.org/devel/tech-ctte`) for more information).

If you take over an old package, you probably want to be listed as the package's official maintainer in the bug system. This will happen automatically once you upload a new version with an updated `Maintainer:` field, although it can take a few hours after the upload is done. If you do not expect to upload a new version for a while, you can use 'The Package Tracking System' on page 25 to get the bug reports. However, make sure that the old maintainer has no problem with the fact that they will continue to receive the bugs during that time.

5.10 Porting and being ported

Debian supports an ever-increasing number of architectures. Even if you are not a porter, and you don't use any architecture but one, it is part of your duty as a maintainer to be aware of issues of portability. Therefore, even if you are not a porter, you should read most of this chapter.

Porting is the act of building Debian packages for architectures that are different from the original architecture of the package maintainer's binary package. It is a unique and essential activity. In fact, porters do most of the actual compiling of Debian packages. For instance, for a single *i386* binary package, there must be a recompile for each architecture, which amounts to 12 more builds.

5.10.1 Being kind to porters

Porters have a difficult and unique task, since they are required to deal with a large volume of packages. Ideally, every source package should build right out of the box. Unfortunately, this is often not the case. This section contains a checklist of "gotchas" often committed by Debian maintainers — common problems which often stymie porters, and make their jobs unnecessarily difficult.

The first and most important thing is to respond quickly to bug or issues raised by porters. Please treat porters with courtesy, as if they were in fact co-maintainers of your package (which, in a way, they are). Please be tolerant of succinct or even unclear bug reports; do your best to hunt down whatever the problem is.

By far, most of the problems encountered by porters are caused by *packaging bugs* in the source packages. Here is a checklist of things you should check or be aware of.

1 Make sure that your `Build-Depends` and `Build-Depends-Indep` settings in `debian/control` are set properly. The best way to validate this is to use the `debootstrap` package to create an unstable chroot environment (see '`debootstrap`' on page 103). Within that chrooted environment, install the `build-essential` package and any package dependencies mentioned in `Build-Depends` and/or `Build-Depends-Indep`. Finally, try building your package within that chrooted environment. These steps can be automated by the use of the `pbuilder` program which is provided by the package of the same name (see '`pbuilder`' on page 103).

   If you can't set up a proper chroot, `dpkg-depcheck` may be of assistance (see '`dpkg-depcheck`' on page 105).

   See the Debian Policy Manual (`http://www.debian.org/doc/debian-policy/`) for instructions on setting build dependencies.

2 Don't set architecture to a value other than "all" or "any" unless you really mean it. In too many cases, maintainers don't follow the instructions in the Debian Policy Manual (`http://www.debian.org/doc/debian-policy/`). Setting your architecture to "i386" is usually incorrect.

3 Make sure your source package is correct. Do `dpkg-source -x` *`package`*`.dsc` to make sure your source package unpacks properly. Then, in there, try building your package from scratch with `dpkg-buildpackage`.

4 Make sure you don't ship your source package with the `debian/files` or `debian/substvars` files. They should be removed by the 'clean' target of `debian/rules`.

5 Make sure you don't rely on locally installed or hacked configurations or programs. For instance, you should never be calling programs in `/usr/local/bin` or the like. Try not to rely on programs being setup in a special way. Try building your package on another machine, even if it's the same architecture.

6 Don't depend on the package you're building being installed already (a sub-case of the above issue).

7 Don't rely on the compiler being a certain version, if possible. If not, then make sure your build dependencies reflect the restrictions, although you are probably asking for trouble, since different architectures sometimes standardize on different compilers.

8 Make sure your debian/rules contains separate "binary-arch" and "binary-indep" targets, as the Debian Policy Manual requires. Make sure that both targets work independently, that is, that you can call the target without having called the other before. To test this, try to run `dpkg-buildpackage -B`.

5.10.2 Guidelines for porter uploads

If the package builds out of the box for the architecture to be ported to, you are in luck and your job is easy. This section applies to that case; it describes how to build and upload your binary package so that it is properly installed into the archive. If you do have to patch the package in order to get it to compile for the other architecture, you are actually doing a source NMU, so consult 'How to do a NMU' on page 54 instead.

For a porter upload, no changes are being made to the source. You do not need to touch any of the files in the source package. This includes `debian/changelog`.

The way to invoke `dpkg-buildpackage` is as `dpkg-buildpackage -B -m`*`porter-email`*. Of course, set *porter-email* to your email address. This will do a binary-only build of only the architecture-dependent portions of the package, using the 'binary-arch' target in `debian/rules`.

If you are working on a Debian machine for your porting efforts and you need to sign your upload locally for its acceptance in the archive, you can run `debsign` on your `.changes` file to have it signed conveniently, or use the remote signing mode of `dpkg-sig`.

Recompilation or binary-only NMU

Sometimes the initial porter upload is problematic because the environment in which the package was built was not good enough (outdated or obsolete library, bad compiler, ...). Then you may just need to recompile it in an updated environment. However, you have to bump the version number in this case, so that the old bad package can be replaced in the Debian archive (`katie` refuses to install new packages if they don't have a version number greater than the currently available one).

You have to make sure that your binary-only NMU doesn't render the package uninstallable. This could happen when a source package generates arch-dependent and arch-independent packages that depend on each other via $(Source-Version).

Despite the required modification of the changelog, these are called binary-only NMUs — there is no need in this case to trigger all other architectures to consider themselves out of date or requiring recompilation.

Such recompilations require special "magic" version numbering, so that the archive maintenance tools recognize that, even though there is a new Debian version, there is no corresponding source update. If you get this wrong, the archive maintainers will reject your upload (due to lack of corresponding source code).

The "magic" for a recompilation-only NMU is triggered by using a suffix appended to the package version number, following the form b<number>. For instance, if the latest version you are recompiling against was version "2.9-3", your NMU should carry a version of "2.9-3+b1". If the latest version was "3.4+b1" (i.e, a native package with a previous recompilation NMU), your NMU should have a version number of "3.4+b2". [2]

[2] In the past, such NMUs used the third-level number on the Debian part of the revision to denote their

Similar to initial porter uploads, the correct way of invoking `dpkg-buildpackage` is `dpkg-buildpackage -B` to only build the architecture-dependent parts of the package.

When to do a source NMU if you are a porter

Porters doing a source NMU generally follow the guidelines found in 'Non-Maintainer Uploads (NMUs)' on page 53, just like non-porters. However, it is expected that the wait cycle for a porter's source NMU is smaller than for a non-porter, since porters have to cope with a large quantity of packages. Again, the situation varies depending on the distribution they are uploading to. It also varies whether the architecture is a candidate for inclusion into the next stable release; the release managers decide and announce which architectures are candidates.

If you are a porter doing an NMU for 'unstable', the above guidelines for porting should be followed, with two variations. Firstly, the acceptable waiting period — the time between when the bug is submitted to the BTS and when it is OK to do an NMU — is seven days for porters working on the unstable distribution. This period can be shortened if the problem is critical and imposes hardship on the porting effort, at the discretion of the porter group. (Remember, none of this is Policy, just mutually agreed upon guidelines.) For uploads to stable or testing, please coordinate with the appropriate release team first.

Secondly, porters doing source NMUs should make sure that the bug they submit to the BTS should be of severity 'serious' or greater. This ensures that a single source package can be used to compile every supported Debian architecture by release time. It is very important that we have one version of the binary and source package for all architecture in order to comply with many licenses.

Porters should try to avoid patches which simply kludge around bugs in the current version of the compile environment, kernel, or libc. Sometimes such kludges can't be helped. If you have to kludge around compiler bugs and the like, make sure you `#ifdef` your work properly; also, document your kludge so that people know to remove it once the external problems have been fixed.

Porters may also have an unofficial location where they can put the results of their work during the waiting period. This helps others running the port have the benefit of the porter's work, even during the waiting period. Of course, such locations have no official blessing or status, so buyer beware.

5.10.3 Porting infrastructure and automation

There is infrastructure and several tools to help automate package porting. This section contains a brief overview of this automation and porting to these tools; see the package documentation or references for full information.

recompilation-only status; however, this syntax was ambiguous with native packages and did not allow proper ordering of recompile-only NMUs, source NMUs, and security NMUs on the same package, and has therefore been abandoned in favor of this new syntax.

Mailing lists and web pages

Web pages containing the status of each port can be found at `http://www.debian.org/ports/`.

Each port of Debian has a mailing list. The list of porting mailing lists can be found at `http://lists.debian.org/ports.html`. These lists are used to coordinate porters, and to connect the users of a given port with the porters.

Porter tools

Descriptions of several porting tools can be found in 'Porting tools' on page 105.

`buildd`

The `buildd` system is used as a distributed, client-server build distribution system. It is usually used in conjunction with *auto-builders*, which are "slave" hosts which simply check out and attempt to auto-build packages which need to be ported. There is also an email interface to the system, which allows porters to "check out" a source package (usually one which cannot yet be auto-built) and work on it.

`buildd` is not yet available as a package; however, most porting efforts are either using it currently or planning to use it in the near future. The actual automated builder is packaged as `sbuild`, see its description in '`sbuild`' on page 103. The complete `buildd` system also collects a number of as yet unpackaged components which are currently very useful and in use continually, such as `andrea` and `wanna-build`.

Some of the data produced by `buildd` which is generally useful to porters is available on the web at `http://buildd.debian.org/`. This data includes nightly updated information from `andrea` (source dependencies) and `quinn-diff` (packages needing recompilation).

We are quite proud of this system, since it has so many possible uses. Independent development groups can use the system for different sub-flavors of Debian, which may or may not really be of general interest (for instance, a flavor of Debian built with `gcc` bounds checking). It will also enable Debian to recompile entire distributions quickly.

The buildds admins of each arch can be contacted at the mail address $arch@buildd.debian.org.

5.10.4 When your package is *not* portable

Some packages still have issues with building and/or working on some of the architectures supported by Debian, and cannot be ported at all, or not within a reasonable amount of time. An example is a package that is SVGA-specific (only i386), or uses other hardware-specific features not supported on all architectures.

In order to prevent broken packages from being uploaded to the archive, and wasting buildd time, you need to do a few things:

- First, make sure your package *does* fail to build on architectures that it cannot support. There are a few ways to achieve this. The preferred way is to have a small testsuite during build time that will test the functionality, and fail if it doesn't work. This is a good idea anyway, as this will prevent (some) broken uploads on all architectures, and also will allow the package to build as soon as the required functionality is available.

  Additionally, if you believe the list of supported architectures is pretty constant, you should change 'any' to a list of supported architectures in debian/control. This way, the build will fail also, and indicate this to a human reader without actually trying.

- In order to prevent autobuilders from needlessly trying to build your package, it must be included in `packages-arch-specific`, a list used by the `wanna-build` script. The current version is available as `http://cvs.debian.org/srcdep/Packages-arch-specific?cvsroot=dak`; please see the top of the file for whom to contact for changes.

Please note that it is insufficient to only add your package to Packages-arch-specific without making it fail to build on unsupported architectures: A porter or any other person trying to build your package might accidently upload it without noticing it doesn't work. If in the past some binary packages were uploaded on unsupported architectures, request their removal by filing a bug against `ftp.debian.org`

5.11 Non-Maintainer Uploads (NMUs)

Under certain circumstances it is necessary for someone other than the official package maintainer to make a release of a package. This is called a non-maintainer upload, or NMU.

This section handles only source NMUs, i.e. NMUs which upload a new version of the package. For binary-only NMUs by porters or QA members, please see 'Recompilation or binary-only NMU' on page 50. If a buildd builds and uploads a package, that too is strictly speaking a binary NMU. See '`buildd`' on the facing page for some more information.

The main reason why NMUs are done is when a developer needs to fix another developer's package in order to address serious problems or crippling bugs or when the package maintainer is unable to release a fix in a timely fashion.

First and foremost, it is critical that NMU patches to source should be as non-disruptive as possible. Do not do housekeeping tasks, do not change the name of modules or files, do not move directories; in general, do not fix things which are not broken. Keep the patch as small as possible. If things bother you aesthetically, talk to the Debian maintainer, talk to the upstream maintainer, or submit a bug. However, aesthetic changes must *not* be made in a non-maintainer upload.

And please remember the Hippocratic Oath: "Above all, do no harm." It is better to leave a package with an open grave bug than applying a non-functional patch, or one that hides the bug instead of resolving it.

5.11.1 How to do a NMU

NMUs which fix important, serious or higher severity bugs are encouraged and accepted. You should endeavor to reach the current maintainer of the package; they might be just about to upload a fix for the problem, or have a better solution.

NMUs should be made to assist a package's maintainer in resolving bugs. Maintainers should be thankful for that help, and NMUers should respect the decisions of maintainers, and try to personally help the maintainer by their work.

A NMU should follow all conventions, written down in this section. For an upload to testing or unstable, this order of steps is recommended:

- Make sure that the package's bugs that the NMU is meant to address are all filed in the Debian Bug Tracking System (BTS). If they are not, submit them immediately.
- Wait a few days for the response from the maintainer. If you don't get any response, you may want to help them by sending the patch that fixes the bug. Don't forget to tag the bug with the "patch" keyword.
- Wait a few more days. If you still haven't got an answer from the maintainer, send them a mail announcing your intent to NMU the package. Prepare an NMU as described in this section, and test it carefully on your machine (cf. 'Testing the package' on page 32). Double check that your patch doesn't have any unexpected side effects. Make sure your patch is as small and as non-disruptive as it can be.
- Upload your package to incoming in `DELAYED/7-day` (cf. 'Delayed uploads' on page 35), send the final patch to the maintainer via the BTS, and explain to them that they have 7 days to react if they want to cancel the NMU.
- Follow what happens, you're responsible for any bug that you introduced with your NMU. You should probably use 'The Package Tracking System' on page 25 (PTS) to stay informed of the state of the package after your NMU.

At times, the release manager or an organized group of developers can announce a certain period of time in which the NMU rules are relaxed. This usually involves shortening the period during which one is to wait before uploading the fixes, and shortening the DELAYED period. It is important to notice that even in these so-called "bug squashing party" times, the NMU'er has to file bugs and contact the developer first, and act later. Please see 'Bug squashing parties' on page 90 for details.

For the testing distribution, the rules may be changed by the release managers. Please take additional care, and acknowledge that the usual way for a package to enter testing is through unstable.

For the stable distribution, please take extra care. Of course, the release managers may also change the rules here. Please verify before you upload that all your changes are OK for inclusion into the next stable release by the release manager.

When a security bug is detected, the security team may do an NMU, using their own rules. Please refer to 'Handling security-related bugs' on page 41 for more information.

For the differences for Porters NMUs, please see 'When to do a source NMU if you are a porter' on page 51.

Of course, it is always possible to agree on special rules with a maintainer (like the maintainer asking "please upload this fix directly for me, and no diff required").

5.11.2 NMU version numbering

Whenever you have made a change to a package, no matter how trivial, the version number needs to change. This enables our packing system to function.

If you are doing a non-maintainer upload (NMU), you should add a new minor version number to the *debian-revision* part of the version number (the portion after the last hyphen). This extra minor number will start at '1'. For example, consider the package 'foo', which is at version 1.1-3. In the archive, the source package control file would be `foo_1.1-3.dsc`. The upstream version is '1.1' and the Debian revision is '3'. The next NMU would add a new minor number '.1' to the Debian revision; the new source control file would be `foo_1.1-3.1.dsc`.

The Debian revision minor number is needed to avoid stealing one of the package maintainer's version numbers, which might disrupt their work. It also has the benefit of making it visually clear that a package in the archive was not made by the official maintainer.

If there is no *debian-revision* component in the version number then one should be created, starting at '0.1' (but in case of a debian native package still upload it as native package). If it is absolutely necessary for someone other than the usual maintainer to make a release based on a new upstream version then the person making the release should start with the *debian-revision* value '0.1'. The usual maintainer of a package should start their *debian-revision* numbering at '1'.

If you upload a package to testing or stable, sometimes, you need to "fork" the version number tree. For this, version numbers like 1.1-3sarge0.1 could be used.

5.11.3 Source NMUs must have a new changelog entry

Anyone who is doing a source NMU must create a changelog entry, describing which bugs are fixed by the NMU, and generally why the NMU was required and what it fixed. The changelog entry will have the email address of the person who uploaded it in the log entry and the NMU version number in it.

By convention, source NMU changelog entries start with the line

```
* Non-maintainer upload
```

5.11.4 Source NMUs and the Bug Tracking System

Maintainers other than the official package maintainer should make as few changes to the package as possible, and they should always send a patch as a unified context diff (`diff -u`) detailing their changes to the Bug Tracking System.

What if you are simply recompiling the package? If you just need to recompile it for a single architecture, then you may do a binary-only NMU as described in 'Recompilation or binary-only NMU' on page 50 which doesn't require any patch to be sent. If you want the package to be recompiled for all architectures, then you do a source NMU as usual and you will have to send a patch.

Bugs fixed by source NMUs used to be tagged fixed instead of closed, but since version tracking is in place, such bugs are now also closed with the NMU version.

Also, after doing an NMU, you have to send the information to the existing bugs that are fixed by your NMU, including the unified diff. Historically, it was custom to open a new bug and include a patch showing all the changes you have made. The normal maintainer will either apply the patch or employ an alternate method of fixing the problem. Sometimes bugs are fixed independently upstream, which is another good reason to back out an NMU's patch. If the maintainer decides not to apply the NMU's patch but to release a new version, the maintainer needs to ensure that the new upstream version really fixes each problem that was fixed in the non-maintainer release.

In addition, the normal maintainer should *always* retain the entry in the changelog file documenting the non-maintainer upload – and of course, also keep the changes. If you revert some of the changes, please reopen the relevant bug reports.

5.11.5 Building source NMUs

Source NMU packages are built normally. Pick a distribution using the same rules as found in 'Picking a distribution' on page 34, follow the other instructions in 'Uploading a package' on page 35.

Make sure you do *not* change the value of the maintainer in the `debian/control` file. Your name as given in the NMU entry of the `debian/changelog` file will be used for signing the changes file.

5.11.6 Acknowledging an NMU

If one of your packages has been NMU'ed, you have to incorporate the changes in your copy of the sources. This is easy, you just have to apply the patch that has been sent to you. Once this is done, you have to close the bugs that have been tagged fixed by the NMU. The easiest way is to use the `-v` option of `dpkg-buildpackage`, as this allows you to include just all changes since your last maintainer upload. Alternatively, you can close them manually by sending the required mails to the BTS or by adding the required `closes: #nnnn` in the changelog entry of your next upload.

In any case, you should not be upset by the NMU. An NMU is not a personal attack against the maintainer. It is a proof that someone cares enough about the package that they were willing to help you in your work, so you should be thankful. You may also want to ask them if they would be interested in helping you on a more frequent basis as co-maintainer or backup maintainer (see 'Collaborative maintenance' on the following page).

5.11.7 NMU vs QA uploads

Unless you know the maintainer is still active, it is wise to check the package to see if it has been orphaned. The current list of orphaned packages which haven't had their maintainer set correctly is available at `http://qa.debian.org/orphaned.html`. If you perform an NMU on an improperly orphaned package, please set the maintainer to "Debian QA Group <packages@qa.debian.org>".

5.11.8 Who can do an NMU

Only official, registered Debian Developers can do binary or source NMUs. A Debian Developer is someone who has their key in the Debian key ring. Non-developers, however, are encouraged to download the source package and start hacking on it to fix problems; however, rather than doing an NMU, they should just submit worthwhile patches to the Bug Tracking System. Maintainers almost always appreciate quality patches and bug reports.

5.11.9 Terminology

There are two new terms used throughout this section: "binary-only NMU" and "source NMU". These terms are used with specific technical meaning throughout this document. Both binary-only and source NMUs are similar, since they involve an upload of a package by a developer who is not the official maintainer of that package. That is why it's a *non-maintainer* upload.

A source NMU is an upload of a package by a developer who is not the official maintainer, for the purposes of fixing a bug in the package. Source NMUs always involves changes to the source (even if it is just a change to `debian/changelog`). This can be either a change to the upstream source, or a change to the Debian bits of the source. Note, however, that source NMUs may also include architecture-dependent packages, as well as an updated Debian diff.

A binary-only NMU is a recompilation and upload of a binary package for a given architecture. As such, it is usually part of a porting effort. A binary-only NMU is a non-maintainer uploaded binary version of a package, with no source changes required. There are many cases where porters must fix problems in the source in order to get them to compile for their target architecture; that would be considered a source NMU rather than a binary-only NMU. As you can see, we don't distinguish in terminology between porter NMUs and non-porter NMUs.

Both classes of NMUs, source and binary-only, can be lumped under the term "NMU". However, this often leads to confusion, since most people think "source NMU" when they think

"NMU". So it's best to be careful: always use "binary NMU" or "binNMU" for binary-only NMUs.

5.12 Collaborative maintenance

"Collaborative maintenance" is a term describing the sharing of Debian package maintenance duties by several people. This collaboration is almost always a good idea, since it generally results in higher quality and faster bug fix turnaround times. It is strongly recommended that packages with a priority of `Standard` or which are part of the base set have co-maintainers.

Generally there is a primary maintainer and one or more co-maintainers. The primary maintainer is the person whose name is listed in the `Maintainer` field of the `debian/control` file. Co-maintainers are all the other maintainers.

In its most basic form, the process of adding a new co-maintainer is quite easy:

- Setup the co-maintainer with access to the sources you build the package from. Generally this implies you are using a network-capable version control system, such as `CVS` or `Subversion`. Alioth (see 'Debian's GForge installation: Alioth' on page 29) provides such tools, amongst others.
- Add the co-maintainer's correct maintainer name and address to the `Uploaders` field in the global part of the `debian/control` file.

  ```
  Uploaders: John Buzz <jbuzz@debian.org>, Adam Rex <arex@debian.org>
  ```

- Using the PTS ('The Package Tracking System' on page 25), the co-maintainers should subscribe themselves to the appropriate source package.

Another form of collaborative maintenance is team maintenance, which is recommended if you maintain several packages with the same group of developers. In that case, the Maintainer and Uploaders field of each package must be managed with care. It is recommended to choose between one of the two following schemes:

1. Put the team member mainly responsible for the package in the Maintainer field. In the Uploaders, put the mailing list address, and the team members who care for the package.
2. Put the mailing list address in the Maintainer field. In the Uploaders field, put the team members who care for the package. In this case, you must make sure the mailing list accept bug reports without any human interaction (like moderation for non-subscribers).

In any case, it is a bad idea to automatically put all team members in the Uploaders field. It clutters the Developer's Package Overview listing (see 'Developer's packages overview' on page 29) with packages one doesn't really care for, and creates a false sense of good maintenance.

5.13 The testing distribution

5.13.1 Basics

Packages are usually installed into the 'testing' distribution after they have undergone some degree of testing in unstable.

They must be in sync on all architectures and mustn't have dependencies that make them uninstallable; they also have to have generally no known release-critical bugs at the time they're installed into testing. This way, 'testing' should always be close to being a release candidate. Please see below for details.

5.13.2 Updates from unstable

The scripts that update the *testing* distribution are run each day after the installation of the updated packages; these scripts are called *britney*. They generate the `Packages` files for the *testing* distribution, but they do so in an intelligent manner; they try to avoid any inconsistency and to use only non-buggy packages.

The inclusion of a package from *unstable* is conditional on the following:

- The package must have been available in *unstable* for 2, 5 or 10 days, depending on the urgency (high, medium or low). Please note that the urgency is sticky, meaning that the highest urgency uploaded since the previous testing transition is taken into account. Those delays may be doubled during a freeze, or testing transitions may be switched off altogether;
- It must have the same number or fewer release-critical bugs than the version currently available in *testing*;
- It must be available on all architectures on which it has previously been built in unstable. 'The `madison` utility' on page 24 may be of interest to check that information;
- It must not break any dependency of a package which is already available in *testing*;
- The packages on which it depends must either be available in *testing* or they must be accepted into *testing* at the same time (and they will be if they fulfill all the necessary criteria);

To find out whether a package is progressing into testing or not, see the testing script output on the web page of the testing distribution (`http://www.debian.org/devel/testing`), or use the program `grep-excuses` which is in the `devscripts` package. This utility can easily be used in a `crontab(5)` to keep yourself informed of the progression of your packages into *testing*.

The `update_excuses` file does not always give the precise reason why the package is refused; you may have to find it on your own by looking for what would break with the inclusion of the

package. The testing web page (http://www.debian.org/devel/testing) gives some more information about the usual problems which may be causing such troubles.

Sometimes, some packages never enter *testing* because the set of inter-relationship is too complicated and cannot be sorted out by the scripts. See below for details.

Some further dependency analysis is shown on http://bjorn.haxx.se/debian/ — but be warned, this page also shows build dependencies which are not considered by britney.

out-of-date

For the testing migration script, "outdated" means: There are different versions in unstable for the release architectures (except for the architectures in fuckedarches; fuckedarches is a list of architectures that don't keep up (in update_out.py), but currently, it's empty). "outdated" has nothing whatsoever to do with the architectures this package has in testing.

Consider this example:

```
foo      | alpha | arm
---------+-------+----
testing  |   1   |  -
unstable |   1   |  2
```

The package is out of date on alpha in unstable, and will not go to testing. And removing foo from testing would not help at all, the package is still out of date on alpha, and will not propagate to testing.

However, if ftp-master removes a package in unstable (here on arm):

```
foo      | alpha | arm | hurd-i386
---------+-------+-----+----------
testing  |   1   |  1  |    -
unstable |   2   |  -  |    1
```

In this case, the package is up to date on all release architectures in unstable (and the extra hurd-i386 doesn't matter, as it's not a release architecture).

Sometimes, the question is raised if it is possible to allow packages in that are not yet built on all architectures: No. Just plainly no. (Except if you maintain glibc or so.)

Removals from testing

Sometimes, a package is removed to allow another package in: This happens only to allow *another* package to go in if it's ready in every other sense. Suppose e.g. that *a* cannot be installed with the new version of *b*; then *a* may be removed to allow *b* in.

Of course, there is another reason to remove a package from testing: It's just too buggy (and having a single RC-bug is enough to be in this state).

Furthermore, if a package has been removed from unstable, and no package in testing depends on it any more, then it will automatically be removed.

circular dependencies

A situation which is not handled very well by britney is if package *a* depends on the new version of package *b*, and vice versa.

An example of this is:

```
  | testing         |  unstable
--+-----------------+------------
a | 1; depends: b=1 |  2; depends: b=2
b | 1; depends: a=1 |  2; depends: a=2
```

Neither package *a* nor package *b* is considered for update.

Currently, this requires some manual hinting from the release team. Please contact them by sending mail to <debian-release@lists.debian.org> if this happens to one of your packages.

influence of package in testing

Generally, there is nothing that the status of a package in testing means for transition of the next version from unstable to testing, with two exceptions: If the RC-bugginess of the package goes down, it may go in even if it is still RC-buggy. The second exception is if the version of the package in testing is out of sync on the different arches: Then any arch might just upgrade to the version of the source package; however, this can happen only if the package was previously forced through, the arch is in fuckedarches, or there was no binary package of that arch present in unstable at all during the testing migration.

In summary this means: The only influence that a package being in testing has on a new version of the same package is that the new version might go in easier.

details

If you are interested in details, this is how britney works:

The packages are looked at to determine whether they are valid candidates. This gives the "update excuses". The most common reasons why a package is not considered are too young, RC-bugginess, and out of date on some arches. For this part of britney, the release managers have hammers of various sizes to force britney to consider a package. (Also, the base freeze is

coded in that part of britney.) (There is a similar thing for binary-only updates, but this is not described here. If you're interested in that, please peruse the code.)

Now, the more complex part happens: Britney tries to update testing with the valid candidates; first, each package alone, and then larger and even larger sets of packages together. Each try is accepted if testing is not more uninstallable after the update than before. (Before and after this part, some hints are processed; but as only release masters can hint, this is probably not so important for you.)

If you want to see more details, you can look it up on merkel:/org/ftp.debian.org/testing/update_out/ (or there in ~aba/testing/update_out to see a setup with a smaller packages file). Via web, it's at `http://ftp-master.debian.org/testing/update_out_code/`

The hints are available via `http://ftp-master.debian.org/testing/hints/`.

5.13.3 Direct updates to testing

The testing distribution is fed with packages from unstable according to the rules explained above. However, in some cases, it is necessary to upload packages built only for testing. For that, you may want to upload to *testing-proposed-updates*.

Keep in mind that packages uploaded there are not automatically processed, they have to go through the hands of the release manager. So you'd better have a good reason to upload there. In order to know what a good reason is in the release managers' eyes, you should read the instructions that they regularly give on `<debian-devel-announce@lists.debian.org>`.

You should not upload to *testing-proposed-updates* when you can update your packages through *unstable*. If you can't (for example because you have a newer development version in unstable), you may use this facility, but it is recommended that you ask for authorization from the release manager first. Even if a package is frozen, updates through unstable are possible, if the upload via unstable does not pull in any new dependencies.

Version numbers are usually selected by adding the codename of the testing distribution and a running number, like 1.2sarge1 for the first upload through testing-proposed-updates of package version 1.2.

Please make sure you didn't miss any of these items in your upload:

- Make sure that your package really needs to go through *testing-proposed-updates*, and can't go through unstable;
- Make sure that you included only the minimal amount of changes;
- Make sure that you included an appropriate explanation in the changelog;
- Make sure that you've written *testing* or *testing-proposed-updates* into your target distribution;

- Make sure that you've built and tested your package in *testing*, not in *unstable*;
- Make sure that your version number is higher than the version in *testing* and *testing-proposed-updates*, and lower than in *unstable*;
- After uploading and successful build on all platforms, contact the release team at `<debian-release@lists.debian.org>` and ask them to approve your upload.

5.13.4 Frequently asked questions

What are release-critical bugs, and how do they get counted?

All bugs of some higher severities are by default considered release-critical; currently, these are critical, grave, and serious bugs.

Such bugs are presumed to have an impact on the chances that the package will be released with the stable release of Debian: in general, if a package has open release-critical bugs filed on it, it won't get into "testing", and consequently won't be released in "stable".

The unstable bug count are all release-critical bugs without either any release-tag (such as potato, woody) or with release-tag sid; also, only if they are neither fixed nor set to sarge-ignore. The "testing" bug count for a package is considered to be roughly the bug count of unstable count at the last point when the "testing" version equalled the "unstable" version.

This will change post-sarge, as soon as we have versions in the bug tracking system.

How could installing a package into "testing" possibly break other packages?

The structure of the distribution archives is such that they can only contain one version of a package; a package is defined by its name. So when the source package acmefoo is installed into "testing", along with its binary packages acme-foo-bin, acme-bar-bin, libacme-foo1 and libacme-foo-dev, the old version is removed.

However, the old version may have provided a binary package with an old soname of a library, such as libacme-foo0. Removing the old acmefoo will remove libacme-foo0, which will break any packages which depend on it.

Evidently, this mainly affects packages which provide changing sets of binary packages in different versions (in turn, mainly libraries). However, it will also affect packages upon which versioned dependencies have been declared of the ==, <=, or << varieties.

When the set of binary packages provided by a source package change in this way, all the packages that depended on the old binaries will have to be updated to depend on the new binaries instead. Because installing such a source package into "testing" breaks all the packages that depended on it in "testing", some care has to be taken now: all the depending packages must be updated and ready to be installed themselves so that they won't be broken, and, once everything is ready, manual intervention by the release manager or an assistant is normally required.

If you are having problems with complicated groups of packages like this, contact debian-devel or debian-release for help.

Chapter 6

Best Packaging Practices

Debian's quality is largely due to the Debian Policy (http://www.debian.org/doc/debian-policy/), which defines explicit baseline requirements which all Debian packages must fulfill. Yet there is also a shared history of experience which goes beyond the Debian Policy, an accumulation of years of experience in packaging. Many very talented people have created great tools, tools which help you, the Debian maintainer, create and maintain excellent packages.

This chapter provides some best practices for Debian developers. All recommendations are merely that, and are not requirements or policy. These are just some subjective hints, advice and pointers collected from Debian developers. Feel free to pick and choose whatever works best for you.

6.1 Best practices for `debian/rules`

The following recommendations apply to the `debian/rules` file. Since `debian/rules` controls the build process and selects the files which go into the package (directly or indirectly), it's usually the file maintainers spend the most time on.

6.1.1 Helper scripts

The rationale for using helper scripts in `debian/rules` is that they let maintainers use and share common logic among many packages. Take for instance the question of installing menu entries: you need to put the file into `/usr/lib/menu` (or `/usr/lib/menu` for executable binary menufiles, if this is needed), and add commands to the maintainer scripts to register and unregister the menu entries. Since this is a very common thing for packages to do, why should each maintainer rewrite all this on their own, sometimes with bugs? Also, supposing the menu directory changed, every package would have to be changed.

Helper scripts take care of these issues. Assuming you comply with the conventions expected by the helper script, the helper takes care of all the details. Changes in policy can be made in

the helper script; then packages just need to be rebuilt with the new version of the helper and no other changes.

'Overview of Debian Maintainer Tools' on page 99 contains a couple of different helpers. The most common and best (in our opinion) helper system is `debhelper`. Previous helper systems, such as `debmake`, were "monolithic": you couldn't pick and choose which part of the helper you found useful, but had to use the helper to do everything. `debhelper`, however, is a number of separate little `dh_*` programs. For instance, `dh_installman` installs and compresses man pages, `dh_installmenu` installs menu files, and so on. Thus, it offers enough flexibility to be able to use the little helper scripts, where useful, in conjunction with handcrafted commands in `debian/rules`.

You can get started with `debhelper` by reading `debhelper(1)`, and looking at the examples that come with the package. `dh_make`, from the `dh-make` package (see '`dh-make`' on page 102), can be used to convert a "vanilla" source package to a `debhelper`ized package. This shortcut, though, should not convince you that you do not need to bother understanding the individual `dh_*` helpers. If you are going to use a helper, you do need to take the time to learn to use that helper, to learn its expectations and behavior.

Some people feel that vanilla `debian/rules` files are better, since you don't have to learn the intricacies of any helper system. This decision is completely up to you. Use what works for you. Many examples of vanilla `debian/rules` files are available at `http://arch.debian.org/arch/private/srivasta/`.

6.1.2 Separating your patches into multiple files

Big, complex packages may have many bugs that you need to deal with. If you correct a number of bugs directly in the source, and you're not careful, it can get hard to differentiate the various patches that you applied. It can get quite messy when you have to update the package to a new upstream version which integrates some of the fixes (but not all). You can't take the total set of diffs (e.g., from `.diff.gz`) and work out which patch sets to back out as a unit as bugs are fixed upstream.

Unfortunately, the packaging system as such currently doesn't provide for separating the patches into several files. Nevertheless, there are ways to separate patches: the patch files are shipped within the Debian patch file (`.diff.gz`), usually within the `debian/` directory. The only difference is that they aren't applied immediately by dpkg-source, but by the `build` rule of `debian/rules`. Conversely, they are reverted in the `clean` rule.

`dbs` is one of the more popular approaches to this. It does all of the above, and provides a facility for creating new and updating old patches. See the package `dbs` for more information and `hello-dbs` for an example.

`dpatch` also provides these facilities, but it's intended to be even easier to use. See the package `dpatch` for documentation and examples (in `/usr/share/doc/dpatch`).

6.1.3 Multiple binary packages

A single source package will often build several binary packages, either to provide several flavors of the same software (e.g., the `vim` source package) or to make several small packages instead of a big one (e.g., so the user can install only the subset needed, and thus save some disk space).

The second case can be easily managed in `debian/rules`. You just need to move the appropriate files from the build directory into the package's temporary trees. You can do this using `install` or `dh_install` from `debhelper`. Be sure to check the different permutations of the various packages, ensuring that you have the inter-package dependencies set right in `debian/control`.

The first case is a bit more difficult since it involves multiple recompiles of the same software but with different configuration options. The `vim` source package is an example of how to manage this using an hand-crafted `debian/rules` file.

6.2 Best practices for `debian/control`

The following practices are relevant to the `debian/control` file. They supplement the Policy on package descriptions (http://www.debian.org/doc/debian-policy/ch-binary.html#s-descriptions).

The description of the package, as defined by the corresponding field in the `control` file, contains both the package synopsis and the long description for the package. 'General guidelines for package descriptions' on the current page describes common guidelines for both parts of the package description. Following that, 'The package synopsis, or short description' on the following page provides guidelines specific to the synopsis, and 'The long description' on page 69 contains guidelines specific to the description.

6.2.1 General guidelines for package descriptions

The package description should be written for the average likely user, the average person who will use and benefit from the package. For instance, development packages are for developers, and can be technical in their language. More general-purpose applications, such as editors, should be written for a less technical user.

Our review of package descriptions lead us to conclude that most package descriptions are technical, that is, are not written to make sense for non-technical users. Unless your package really is only for technical users, this is a problem.

How do you write for non-technical users? Avoid jargon. Avoid referring to other applications or frameworks that the user might not be familiar with — "GNOME" or "KDE" is fine, since users are probably familiar with these terms, but "GTK+" is probably not. Try not to assume any knowledge at all. If you must use technical terms, introduce them.

Be objective. Package descriptions are not the place for advocating your package, no matter how much you love it. Remember that the reader may not care about the same things you care about.

References to the names of any other software packages, protocol names, standards, or specifications should use their canonical forms, if one exists. For example, use "X Window System", "X11", or "X"; not "X Windows", "X-Windows", or "X Window". Use "GTK+", not "GTK" or "gtk". Use "GNOME", not "Gnome". Use "PostScript", not "Postscript" or "postscript".

If you are having problems writing your description, you may wish to send it along to `<debian-l10n-english@lists.debian.org>` and request feedback.

6.2.2 The package synopsis, or short description

The synopsis line (the short description) should be concise. It must not repeat the package's name (this is policy).

It's a good idea to think of the synopsis as an appositive clause, not a full sentence. An appositive clause is defined in WordNet as a grammatical relation between a word and a noun phrase that follows, e.g., "Rudolph the red-nosed reindeer". The appositive clause here is "red-nosed reindeer". Since the synopsis is a clause, rather than a full sentence, we recommend that it neither start with a capital nor end with a full stop (period). It should also not begin with an article, either definite ("the") or indefinite ("a" or "an").

It might help to imagine that the synopsis is combined with the package name in the following way:

package-name is a *synopsis*.

Alternatively, it might make sense to think of it as

package-name is *synopsis*.

or, if the package name itself is a plural (such as "developers-tools")

package-name are *synopsis*.

This way of forming a sentence from the package name and synopsis should be considered as a heuristic and not a strict rule. There are some cases where it doesn't make sense to try to form a sentence.

6.2.3 The long description

The long description is the primary information available to the user about a package before they install it. It should provide all the information needed to let the user decide whether to install the package. Assume that the user has already read the package synopsis.

The long description should consist of full and complete sentences.

The first paragraph of the long description should answer the following questions: what does the package do? what task does it help the user accomplish? It is important to describe this in a non-technical way, unless of course the audience for the package is necessarily technical.

The following paragraphs should answer the following questions: Why do I as a user need this package? What other features does the package have? What outstanding features and deficiencies are there compared to other packages (e.g., "if you need X, use Y instead")? Is this package related to other packages in some way that is not handled by the package manager (e.g., "this is the client for the foo server")?

Be careful to avoid spelling and grammar mistakes. Ensure that you spell-check it. Both `ispell` and `aspell` have special modes for checking `debian/control` files:

```
ispell -d american -g debian/control

aspell -d en -D -c debian/control
```

Users usually expect these questions to be answered in the package description:

- What does the package do? If it is an add-on to another package, then the short description of the package we are an add-on to should be put in here.
- Why should I want this package? This is related to the above, but not the same (this is a mail user agent; this is cool, fast, interfaces with PGP and LDAP and IMAP, has features X, Y, and Z).
- If this package should not be installed directly, but is pulled in by another package, this should be mentioned.
- If the package is experimental, or there are other reasons it should not be used, if there are other packages that should be used instead, it should be here as well.
- How is this package different from the competition? Is it a better implementation? more features? different features? Why should I choose this package.

6.2.4 Upstream home page

We recommend that you add the URL for the package's home page in the `Homepage` field of the `Source` section in `debian/control`. Adding this information in the package description itself is considered deprecated.

6.2.5 Version Control System location

There are additional fields for the location of the Version Control System in `debian/control`.

Vcs-Browser

Value of this field should be a `http://` URL pointing to a web-browsable copy of the Version Control System repository used to maintain the given package, if available.

The information is meant to be useful for the final user, willing to browse the latest work done on the package (e.g. when looking for the patch fixing a bug tagged as `pending` in the bug tracking system).

Vcs-*

Value of this field should be a string identifying unequivocally the location of the Version Control System repository used to maintain the given package, if available. * identify the Version Control System; currently the following systems are supported by the package tracking system: `arch`, `bzr` (Bazaar), `cvs`, `darcs`, `git`, `hg` (Mercurial), `mtn` (Monotone), `svn` (Subversion). It is allowed to specify different VCS fields for the same package: they will all be shown in the PTS web interface.

The information is meant to be useful for a user knowledgeable in the given Version Control System and willing to build the current version of a package from the VCS sources. Other uses of this information might include automatic building of the latest VCS version of the given package. To this end the location pointed to by the field should better be version agnostic and point to the main branch (for VCSs supporting such a concept). Also, the location pointed to should be accessible to the final user; fulfilling this requirement might imply pointing to an anonymous access of the repository instead of pointing to an SSH-accessible version of the same.

In the following example, an instance of the field for a Subversion repository of the `vim` package is shown. Note how the URL is in the `svn://` scheme (instead of `svn+ssh://`) and how it points to the `trunk/` branch. The use of the `Vcs-Browser` and `Homepage` fields described above is also shown.

```
Source: vim
Section: editors
Priority: optional
<snip>
Vcs-Svn: svn://svn.debian.org/svn/pkg-vim/trunk/packages/vim
Vcs-Browser: http://svn.debian.org/wsvn/pkg-vim/trunk/packages/vim
Homepage: http://www.vim.org
```

6.3 Best practices for `debian/changelog`

The following practices supplement the Policy on changelog files (http://www.debian.org/doc/debian-policy/ch-docs.html#s-changelogs).

6.3.1 Writing useful changelog entries

The changelog entry for a package revision documents changes in that revision, and only them. Concentrate on describing significant and user-visible changes that were made since the last version.

Focus on *what* was changed — who, how and when are usually less important. Having said that, remember to politely attribute people who have provided notable help in making the package (e.g., those who have sent in patches).

There's no need to elaborate the trivial and obvious changes. You can also aggregate several changes in one entry. On the other hand, don't be overly terse if you have undertaken a major change. Be especially clear if there are changes that affect the behaviour of the program. For further explanations, use the `README.Debian` file.

Use common English so that the majority of readers can comprehend it. Avoid abbreviations, "tech-speak" and jargon when explaining changes that close bugs, especially for bugs filed by users that did not strike you as particularly technically savvy. Be polite, don't swear.

It is sometimes desirable to prefix changelog entries with the names of the files that were changed. However, there's no need to explicitly list each and every last one of the changed files, especially if the change was small or repetitive. You may use wildcards.

When referring to bugs, don't assume anything. Say what the problem was, how it was fixed, and append the "closes: #nnnnn" string. See 'When bugs are closed by new uploads' on page 40 for more information.

6.3.2 Common misconceptions about changelog entries

The changelog entries should **not** document generic packaging issues ("Hey, if you're looking for foo.conf, it's in /etc/blah/."), since administrators and users are supposed to be at least remotely acquainted with how such things are generally arranged on Debian systems. Do, however, mention if you change the location of a configuration file.

The only bugs closed with a changelog entry should be those that are actually fixed in the same package revision. Closing unrelated bugs in the changelog is bad practice. See 'When bugs are closed by new uploads' on page 40.

The changelog entries should **not** be used for random discussion with bug reporters ("I don't see segfaults when starting foo with option bar; send in more info"), general statements on life, the universe and everything ("sorry this upload took me so long, but I caught the flu"), or pleas for help ("the bug list on this package is huge, please lend me a hand"). Such things

usually won't be noticed by their target audience, but may annoy people who wish to read information about actual changes in the package. See 'Responding to bugs' on page 38 for more information on how to use the bug tracking system.

It is an old tradition to acknowledge bugs fixed in non-maintainer uploads in the first changelog entry of the proper maintainer upload. As we have version tracking now, it is enough to keep the NMUed changelog entries and just mention this fact in your own changelog entry.

6.3.3 Common errors in changelog entries

The following examples demonstrate some common errors or examples of bad style in changelog entries.

```
* Fixed all outstanding bugs.
```

This doesn't tell readers anything too useful, obviously.

```
* Applied patch from Jane Random.
```

What was the patch about?

```
* Late night install target overhaul.
```

Overhaul which accomplished what? Is the mention of late night supposed to remind us that we shouldn't trust that code?

```
* Fix vsync FU w/ ancient CRTs.
```

Too many acronyms, and it's not overly clear what the, uh, fsckup (oops, a curse word!) was actually about, or how it was fixed.

```
* This is not a bug, closes: #nnnnnn.
```

First of all, there's absolutely no need to upload the package to convey this information; instead, use the bug tracking system. Secondly, there's no explanation as to why the report is not a bug.

```
* Has been fixed for ages, but I forgot to close; closes: #54321.
```

If for some reason you didn't mention the bug number in a previous changelog entry, there's no problem, just close the bug normally in the BTS. There's no need to touch the changelog file, presuming the description of the fix is already in (this applies to the fixes by the upstream authors/maintainers as well, you don't have to track bugs that they fixed ages ago in your changelog).

```
  * Closes: #12345, #12346, #15432
```

Where's the description? If you can't think of a descriptive message, start by inserting the title of each different bug.

6.3.4 Supplementing changelogs with NEWS.Debian files

Important news about changes in a package can also be put in NEWS.Debian files. The news will be displayed by tools like apt-listchanges, before all the rest of the changelogs. This is the preferred means to let the user know about significant changes in a package. It is better than using debconf notes since it is less annoying and the user can go back and refer to the NEWS.Debian file after the install. And it's better than listing major changes in README.Debian, since the user can easily miss such notes.

The file format is the same as a debian changelog file, but leave off the asterisks and describe each news item with a full paragraph when necessary rather than the more concise summaries that would go in a changelog. It's a good idea to run your file through dpkg-parsechangelog to check its formatting as it will not be automatically checked during build as the changelog is. Here is an example of a real NEWS.Debian file:

```
cron (3.0pl1-74) unstable; urgency=low

    The checksecurity script is no longer included with the cron package:
    it now has its own package, "checksecurity". If you liked the
    functionality provided with that script, please install the new
    package.

 -- Steve Greenland <stevegr@debian.org>  Sat,  6 Sep 2003 17:15:03 -0500
```

The NEWS.Debian file is installed as /usr/share/doc/<package>/NEWS.Debian.gz. It is compressed, and always has that name even in Debian native packages. If you use debhelper, dh_installchangelogs will install debian/NEWS files for you.

Unlike changelog files, you need not update NEWS.Debian files with every release. Only update them if you have something particularly newsworthy that user should know about. If you have no news at all, there's no need to ship a NEWS.Debian file in your package. No news is good news!

6.4 Best practices for maintainer scripts

Maintainer scripts include the files `debian/postinst`, `debian/preinst`, `debian/prerm` and `debian/postrm`. These scripts take care of any package installation or deinstallation setup which isn't handled merely by the creation or removal of files and directories. The following instructions supplement the Debian Policy (http://www.debian.org/doc/debian-policy/).

Maintainer scripts must be idempotent. That means that you need to make sure nothing bad will happen if the script is called twice where it would usually be called once.

Standard input and output may be redirected (e.g. into pipes) for logging purposes, so don't rely on them being a tty.

All prompting or interactive configuration should be kept to a minimum. When it is necessary, you should use the `debconf` package for the interface. Remember that prompting in any case can only be in the `configure` stage of the `postinst` script.

Keep the maintainer scripts as simple as possible. We suggest you use pure POSIX shell scripts. Remember, if you do need any bash features, the maintainer script must have a bash shebang line. POSIX shell or Bash are preferred to Perl, since they enable `debhelper` to easily add bits to the scripts.

If you change your maintainer scripts, be sure to test package removal, double installation, and purging. Be sure that a purged package is completely gone, that is, it must remove any files created, directly or indirectly, in any maintainer script.

If you need to check for the existence of a command, you should use something like

```
if [ -x /usr/sbin/install-docs ]; then ...
```

If you don't wish to hard-code the path of a command in your maintainer script, the following POSIX-compliant shell function may help:

```
pathfind() {
    OLDIFS="$IFS"
    IFS=:
    for p in $PATH; do
        if [ -x "$p/$*" ]; then
            IFS="$OLDIFS"
            return 0
        fi
    done
    IFS="$OLDIFS"
    return 1
}
```

You can use this function to search `$PATH` for a command name, passed as an argument. It returns true (zero) if the command was found, and false if not. This is really the most portable way, since `command -v`, `type`, and `which` are not POSIX.

While `which` is an acceptable alternative, since it is from the required `debianutils` package, it's not on the root partition. That is, it's in `/usr/bin` rather than `/bin`, so one can't use it in scripts which are run before the `/usr` partition is mounted. Most scripts won't have this problem, though.

6.5 Configuration management with `debconf`

`Debconf` is a configuration management system which can be used by all the various packaging scripts (`postinst` mainly) to request feedback from the user concerning how to configure the package. Direct user interactions must now be avoided in favor of `debconf` interaction. This will enable non-interactive installations in the future.

Debconf is a great tool but it is often poorly used. Many common mistakes are listed in the `debconf-devel(7)` man page. It is something that you must read if you decide to use debconf. Also, we document some best practices here.

These guidelines include some writing style and typography recommendations, general considerations about debconf usage as well as more specific recommendations for some parts of the distribution (the installation system for instance).

6.5.1 Do not abuse debconf

Since debconf appeared in Debian, it has been widely abused and several criticisms received by the Debian distribution come from debconf abuse with the need of answering a wide bunch of questions before getting any little thing installed.

Keep usage notes to what they belong: the NEWS.Debian, or README.Debian file. Only use notes for important notes which may directly affect the package usability. Remember that notes will always block the install until confirmed or bother the user by email.

Carefully choose the questions priorities in maintainer scripts. See `debconf-devel(7)` for details about priorities. Most questions should use medium and low priorities.

6.5.2 General recommendations for authors and translators

Write correct English

Most Debian package maintainers are not native English speakers. So, writing properly phrased templates may not be easy for them.

Please use (and abuse) `<debian-l10n-english@lists.debian.org>` mailing list. Have your templates proofread.

Badly written templates give a poor image of your package, of your work... or even of Debian itself.

Avoid technical jargon as much as possible. If some terms sound common to you, they may be impossible to understand for others. If you cannot avoid them, try to explain them (use the extended description). When doing so, try to balance between verbosity and simplicity.

Be kind to translators

Debconf templates may be translated. Debconf, along with its sister package `po-debconf` offers a simple framework for getting templates translated by translation teams or even individuals.

Please use gettext-based templates. Install `po-debconf` on your development system and read its documentation ("man po-debconf" is a good start).

Avoid changing templates too often. Changing templates text induces more work to translators which will get their translation "fuzzied". If you plan changes to your original templates, please contact translators. Most active translators are very responsive and getting their work included along with your modified templates will save you additional uploads. If you use gettext-based templates, the translator's name and e-mail addresses are mentioned in the po files headers.

The use of the `podebconf-report-po` from the po-debconf package is highly recommended to warn translators which have incomplete translations and request them for updates.

If in doubt, you may also contact the translation team for a given language (debian-l10n-xxxxx@lists.debian.org), or the `<debian-i18n@lists.debian.org>` mailing list.

Calls for translations posted to `<debian-i18n@lists.debian.org>` with the `debian/po/templates.pot` file attached or referenced in a URL are encouraged. Be sure to mentions in these calls for new translations which languages you have existing translations for, in order to avoid duplicate work.

Unfuzzy complete translations when correcting typos and spelling

When the text of a debconf template is corrected and you are **sure** that the change does **not** affect translations, please be kind to translators and unfuzzy their translations.

If you don't do so, the whole template will not be translated as long as a translator will send you an update.

To **unfuzzy** translations, you can proceed the following way:

1 Put all incomplete PO files out of the way. You can check the completeness by using (needs the `gettext` package installed):

```
for i in debian/po/*po; do echo -n $i: ; msgfmt -o /dev/null
--statistics $i; done
```

2 move all files which report either fuzzy strings to a temporary place. Files which report no fuzzy strings (only translated and untranslated) will be kept in place.

3 now **and now only**, modify the template for the typos and check again that translation are not impacted (typos, spelling errors, sometimes typographical corrections are usually OK)

4 run `debconf-updatepo`. This will fuzzy all strings you modified in translations. You can see this by running the above again

5 use the following command:

```
for i in debian/po/*po; do msgattrib --output-file=$i --clear-fuzzy $i;
```

6 move back to debian/po the files which showed fuzzy strings in the first step

7 run `debconf-updatepo` again

Do not make assumptions about interfaces

Templates text should not make reference to widgets belonging to some debconf interfaces. Sentences like "If you answer Yes..." have no meaning for users of graphical interfaces which use checkboxes for boolean questions.

String templates should also avoid mentioning the default values in their description. First, because this is redundant with the values seen by the users. Also, because these default values may be different from the maintainer choices (for instance, when the debconf database was preseeded).

More generally speaking, try to avoid referring to user actions. Just give facts.

Do not use first person

You should avoid the use of first person ("I will do this..." or "We recommend..."). The computer is not a person and the Debconf templates do not speak for the Debian developers. You should use neutral construction. Those of you who already wrote scientific publications, just write your templates like you would write a scientific paper. However, try using action voice if still possible, like "Enable this if ..." instead of "This can be enabled if ...".

Be gender neutral

The world is made of men and women. Please use gender-neutral constructions in your writing.

6.5.3 Templates fields definition

This part gives some information which is mostly taken from the `debconf-devel(7)` manual page.

Type

string: Results in a free-form input field that the user can type any string into.

password: Prompts the user for a password. Use this with caution; be aware that the password the user enters will be written to debconf's database. You should probably clean that value out of the database as soon as is possible.

boolean: A true/false choice. Remember: true/false, **not yes/no**...

select: A choice between one of a number of values. The choices must be specified in a field named 'Choices'. Separate the possible values with commas and spaces, like this: Choices: yes, no, maybe

multiselect: Like the select data type, except the user can choose any number of items from the choices list (or chose none of them).

note: Rather than being a question per se, this datatype indicates a note that can be displayed to the user. It should be used only for important notes that the user really should see, since debconf will go to great pains to make sure the user sees it; halting the install for them to press a key, and even mailing the note to them in some cases.

text: This type is now considered obsolete: don't use it.

error: This type is designed to handle error messages. It is mostly similar to the "note" type. Frontends may present it differently (for instance, the dialog frontend of cdebconf draws a red screen instead of the usual blue one).

It is recommended to use this type for any message that needs user attention for a correction of any kind.

Description: short and extended description

Template descriptions have two parts: short and extended. The short description is in the "Description:" line of the template.

The short description should be kept short (50 characters or so) so that it may be accomodated by most debconf interfaces. Keeping it short also helps translators, as usually translations tend to end up being longer than the original.

The short description should be able to stand on its own. Some interfaces do not show the long description by default, or only if the user explicitely asks for it or even do not show it at all. Avoid things like "What do you want to do?"

The short description does not necessarily have to be a full sentence. This is part of the "keep it short and efficient" recommendation.

The extended description should not repeat the short description word for word. If you can't think up a long description, then first, think some more. Post to debian-devel. Ask for help. Take a writing class! That extended description is important. If after all that you still can't come up with anything, leave it blank.

The extended description should use complete sentences. Paragraphs should be kept short for improved readability. Do not mix two ideas in the same paragraph but rather use another paragraph.

Don't be too verbose. User tend to ignore too long screens. 20 lines are by experience a border you shouldn't cross, because that means that in the classical dialog interface, people will need to scroll, and lot of people just don't do that.

The extended description should **never** include a question.

For specific rules depending on templates type (string, boolean, etc.), please read below.

Choices

This field should be used for Select and Multiselect types. It contains the possible choices which will be presented to users. These choices should be separated by commas.

Default

This field is optional. It contains the default answer for string, select and multiselect templates. For multiselect templates, it may contain a comma-separated list of choices.

6.5.4 Templates fields specific style guide

Type field

No specific indication except: use the appropriate type by referring to the previous section.

Description field

Below are specific instructions for properly writing the Description (short and extended) depending on the template type.

String/password templates

- The short description is a prompt and **not** a title. Avoid question style prompts ("IP Address?") in favour of "opened" prompts ("IP address:"). The use of colons is recommended.
- The extended description is a complement to the short description. In the extended part, explain what is being asked, rather than ask the same question again using longer words. Use complete sentences. Terse writing style is strongly discouraged.

Boolean templates

- The short description should be phrased in the form of a question which should be kept short and should generally end with a question mark. Terse writing style is permitted and even encouraged if the question is rather long (remember that translations are often longer than original versions)
- Again, please avoid referring to specific interface widgets. A common mistake for such templates is "if you answer Yes"-type constructions.

Select/Multiselect

- The short description is a prompt and **not** a title. Do **not** use useless "Please choose..." constructions. Users are clever enough to figure out they have to choose something...:)
- The extended description will complete the short description. It may refer to the available choices. It may also mention that the user may choose more than one of the available choices, if the template is a multiselect one (although the interface often makes this clear).

Notes

- The short description should be considered to be a *title*.
- The extended description is what will be displayed as a more detailed explanation of the note. Phrases, no terse writing style.
- **Do not abuse debconf.** Notes are the most common way to abuse debconf. As written in debconf-devel manual page: it's best to use them only for warning about very serious problems. The NEWS.Debian or README.Debian files are the appropriate location for a lot of notes. If, by reading this, you consider converting your Note type templates to entries in NEWS/Debian or README.Debian, plus consider keeping existing translations for the future.

Choices field

If the Choices are likely to change often, please consider using the "__Choices" trick. This will split each individual choice into a single string, which will considerably help translators for doing their work.

Default field

If the default value, for a "select" template, is likely to vary depending on the user language (for instance, if the choice is a language choice), please use the "_DefaultChoice" trick.

This special field allow translators to put the most appropriate choice according to their own language. It will become the default choice when their language is used while your own mentioned Default Choice will be used chan using English.

Example, taken from the geneweb package templates:

```
Template: geneweb/lang
Type: select
__Choices: Afrikaans (af), Bulgarian (bg), Catalan (ca), Chinese (zh), Czech
# This is the default choice. Translators may put their own language here
# instead of the default.
# WARNING : you MUST use the ENGLISH FORM of your language
# For instance, the french translator will need to put "French (fr)" here.
_DefaultChoice: English (en)[ translators, please see comment in PO files]
_Description: Geneweb default language:
```

Note the use of brackets which allow internal comments in debconf fields. Also note the use of comments which will show up in files the translators will work with.

The comments are needed as the DefaultChoice trick is a bit confusing: the translators may put their own choice

Default field

Do NOT use empty default field. If you don't want to use default values, do not use Default at all.

If you use po-debconf (and you **should**, see 2.2), consider making this field translatable, if you think it may be translated.

If the default value may vary depending on language/country (for instance the default value for a language choice), consider using the special "_DefaultChoice" type documented in `po-debconf(7)`).

6.6 Internationalization

6.6.1 Handling debconf translations

Like porters, translators have a difficult task. They work on many packages and must collaborate with many different maintainers. Moreover, most of the time, they are not native English speakers, so you may need to be particularly patient with them.

The goal of `debconf` was to make packages configuration easier for maintainers and for users. Originally, translation of debconf templates was handled with `debconf-mergetemplate`. However, that technique is now deprecated; the best way to accomplish `debconf` internationalization is by using the `po-debconf` package. This method is easier both for maintainer and translators; transition scripts are provided.

Using `po-debconf`, the translation is stored in `po` files (drawing from `gettext` translation techniques). Special template files contain the original messages and mark which fields are translatable. When you change the value of a translatable field, by calling `debconf-updatepo`, the translation is marked as needing attention from the translators. Then, at build time, the `dh_installdebconf` program takes care of all the needed magic to add the template along with the up-to-date translations into the binary packages. Refer to the `po-debconf(7)` manual page for details.

6.6.2 Internationalized documentation

Internationalizing documentation is crucial for users, but a lot of labor. There's no way to eliminate all that work, but you can make things easier for translators.

If you maintain documentation of any size, its easier for translators if they have access to a source control system. That lets translators see the differences between two versions of the documentation, so, for instance, they can see what needs to be retranslated. It is recommended that the translated documentation maintain a note about what source control revision the translation is based on. An interesting system is provided by doc-check (`http://cvs.debian.org/boot-floppies/documentation/doc-check?rev=HEADcontent-type=text/vnd.viewcvs-markup`) in the `boot-floppies` package, which shows an overview of the translation status for any given language, using structured comments for the current revision of the file to be translated and, for a translated file, the revision of the original file the translation is based on. You might wish to adapt and provide that in your CVS area.

If you maintain XML or SGML documentation, we suggest that you isolate any language-independent information and define those as entities in a separate file which is included by all the different translations. This makes it much easier, for instance, to keep URLs up to date across multiple files.

6.7 Common packaging situations

6.7.1 Packages using `autoconf/automake`

Keeping `autoconf`'s `config.sub` and `config.guess` files up to date is critical for porters, especially on more volatile architectures. Some very good packaging practices for any package using `autoconf` and/or `automake` have been synthesized in `/usr/share/doc/autotools-dev/README.Debian.gz` from the `autotools-dev` package. You're strongly encouraged to read this file and to follow the given recommendations.

6.7.2 Libraries

Libraries are always difficult to package for various reasons. The policy imposes many constraints to ease their maintenance and to make sure upgrades are as simple as possible when a new upstream version comes out. Breakage in a library can result in dozens of dependent packages breaking.

Good practices for library packaging have been grouped in the library packaging guide (`http://www.netfort.gr.jp/~dancer/column/libpkg-guide/`).

6.7.3 Documentation

Be sure to follow the Policy on documentation (`http://www.debian.org/doc/debian-policy/ch-docs.html`).

If your package contains documentation built from XML or SGML, we recommend you not ship the XML or SGML source in the binary package(s). If users want the source of the documentation, they should retrieve the source package.

Policy specifies that documentation should be shipped in HTML format. We also recommend shipping documentation in PDF and plain text format if convenient and if output of reasonable quality is possible. However, it is generally not appropriate to ship plain text versions of documentation whose source format is HTML.

Major shipped manuals should register themselves with `doc-base` on installation. See the `doc-base` package documentation for more information.

6.7.4 Specific types of packages

Several specific types of packages have special sub-policies and corresponding packaging rules and practices:

- Perl related packages have a Perl policy (`http://www.debian.org/doc/packaging-manuals/perl-policy/`), some examples of packages following that policy are `libdbd-pg-perl` (binary perl module) or `libmldbm-perl` (arch independent perl module).

- Python related packages have their python policy; see `/usr/share/doc/python/python-policy.txt.gz` in the `python` package.
- Emacs related packages have the emacs policy (`http://www.debian.org/doc/packaging-manuals/debian-emacs-policy`).
- Java related packages have their java policy (`http://www.debian.org/doc/packaging-manuals/java-policy/`).
- Ocaml related packages have their own policy, found in `/usr/share/doc/ocaml/ocaml_packaging_policy.gz` from the `ocaml` package. A good example is the `camlzip` source package.
- Packages providing XML or SGML DTDs should conform to the recommendations found in the `sgml-base-doc` package.
- Lisp packages should register themselves with `common-lisp-controller`, about which see `/usr/share/doc/common-lisp-controller/README.packaging`.

6.7.5 Architecture-independent data

It is not uncommon to have a large amount of architecture-independent data packaged with a program. For example, audio files, a collection of icons, wallpaper patterns, or other graphic files. If the size of this data is negligible compared to the size of the rest of the package, it's probably best to keep it all in a single package.

However, if the size of the data is considerable, consider splitting it out into a separate, architecture-independent package ("_all.deb"). By doing this, you avoid needless duplication of the same data into eleven or more .debs, one per each architecture. While this adds some extra overhead into the `Packages` files, it saves a lot of disk space on Debian mirrors. Separating out architecture-independent data also reduces processing time of `lintian` or `linda` (see 'Package lint tools' on page 100) when run over the entire Debian archive.

6.7.6 Needing a certain locale during build

If you need a certain locale during build, you can create a temporary file via this trick:

If you set LOCPATH to the equivalent of /usr/lib/locale, and LC_ALL to the name of the locale you generate, you should get what you want without being root. Something like this:

```
LOCALE_PATH=debian/tmpdir/usr/lib/locale
LOCALE_NAME=en_IN
LOCALE_CHARSET=UTF-8

mkdir -p $LOCALE_PATH
localedef -i "$LOCALE_NAME.$LOCALE_CHARSET" -f "$LOCALE_CHARSET" "$LOCALE_PAT
```

```
# Using the locale
LOCPATH=$LOCALE_PATH LC_ALL=$LOCALE_NAME.$LOCALE_CHARSET date
```

6.7.7 Make transition packages deborphan compliant

Deborphan is a program for helping users to detect which packages can safely be removed from the system, i.e. the ones that have no packages depending on them. The default operation is to search only within the libs and oldlibs sections, to hunt down unused libraries. But when passed the right argument, it tries to catch other useless packages.

For example, with –guess-dummy, deborphan tries to search all transitional packages which were needed for upgrade but which can now safely be removed. For that, it looks for the string "dummy" or "transitional" in their short description.

So, when you are creating such a package, please make sure to add this text to your short description. If you are looking for examples, just run:

```
apt-cache search .|grep dummy
```

or

```
apt-cache search .|grep transitional
```

.

6.7.8 Best practices for `orig.tar.gz` files

There are two kinds of original source tarballs: Pristine source and repackaged upstream source.

Pristine source

The defining characteristic of a pristine source tarball is that the .orig.tar.gz file is byte-for-byte identical to a tarball officially distributed by the upstream author. [1] This makes it possible to use checksums to easily verify that all changes between Debian's version and upstream's are contained in the Debian diff. Also, if the original source is huge, upstream authors and others who already have the upstream tarball can save download time if they want to inspect your packaging in detail.

[1] We cannot prevent upstream authors from changing the tarball they distribute without also incrementing the version number, so there can be no guarantee that a pristine tarball is identical to what upstream *currently* distributing at any point in time. All that can be expected is that it is identical to something that upstream once *did* distribute. If a difference arises later (say, if upstream notices that he wasn't using maximal comression in his original distribution and then re-`gzip`s it), that's just too bad. Since there is no good way to upload a new .orig.tar.gz for the same version, there is not even any point in treating this situation as a bug.

There is no universally accepted guidelines that upstream authors follow regarding to the directory structure inside their tarball, but `dpkg-source` is nevertheless able to deal with most upstream tarballs as pristine source. Its strategy is equivalent to the following:

1 It unpacks the tarball in an empty temporary directory by doing

```
zcat path/to/<packagename>_<upstream-version>.orig.tar.gz | tar xf -
```

2 If, after this, the temporary directory contains nothing but one directory and no other files, `dpkg-source` renames that directory to `<packagename>-<upstream-version>(.orig)`. The name of the top-level directory in the tarball does not matter, and is forgotten.

3 Otherwise, the upstream tarball must have been packaged without a common top-level directory (shame on the upstream author!). In this case, `dpkg-source` renames the temporary directory *itself* to `<packagename>-<upstream-version>(.orig)`.

Repackaged upstream source

You **should** upload packages with a pristine source tarball if possible, but there are various reasons why it might not be possible. This is the case if upstream does not distribute the source as gzipped tar at all, or if upstream's tarball contains non-DFSG-free material that you must remove before uploading.

In these cases the developer must construct a suitable .orig.tar.gz file himself. We refer to such a tarball as a "repackaged upstream source". Note that a "repackaged upstream source" is different from a Debian-native package. A repackaged source still comes with Debian-specific changes in a separate `.diff.gz` and still has a version number composed of `<upstream-version>` and `<debian-revision>`.

There may be cases where it is desirable to repackage the source even though upstream distributes a `.tar.gz` that could in principle be used in its pristine form. The most obvious is if *significant* space savings can be achieved by recompressing the tar archive or by removing genuinely useless cruft from the upstream archive. Use your own discretion here, but be prepared to defend your decision if you repackage source that could have been pristine.

A repackaged .orig.tar.gz

1 **must** contain detailed information how the repackaged source was obtained, and how this can be reproduced in the `debian/copyright`. It is also a good idea to provide a `get-orig-source` target in your `debian/rules` file that repeats the process, as described in the Policy Manual, Main building script: debian/rules (http://www.debian.org/doc/debian-policy/ch-source.html#s-debianrules).

2 **should not** contain any file that does not come from the upstream author(s), or whose contents has been changed by you. [2]

3 **should**, except where impossible for legal reasons, preserve the entire building and portablility infrastructure provided by the upstream author. For example, it is not a sufficient reason for omitting a file that it is used only when building on MS-DOS. Similarly, a Makefile provided by upstream should not be omitted even if the first thing your `debian/rules` does is to overwrite it by running a configure script.

(*Rationale:* It is common for Debian users who need to build software for non-Debian platforms to fetch the source from a Debian mirror rather than trying to locate a canonical upstream distribution point).

4 **should** use `<packagename>-<upstream-version>.orig` as the name of the top-level directory in its tarball. This makes it possible to distinguish pristine tarballs from repackaged ones.

5 **should** be gzipped with maximal compression.

The canonical way to meet the latter two points is to let `dpkg-source -b` construct the repackaged tarball from an unpacked directory.

Changing binary files in `diff.gz`

Sometimes it is necessary to change binary files contained in the original tarball, or to add binary files that are not in it. If this is done by simply copying the files into the debianized source tree, `dpkg-source` will not be able to handle this. On the other hand, according to the guidelines given above, you cannot include such a changed binary file in a repackaged `orig.tar.gz`. Instead, include the file in the `debian` directory in `uuencoded` (or similar) form [3]. The file would then be decoded and copied to its place during the build process. Thus the change will be visible quite easy.

Some packages use `dbs` to manage patches to their upstream source, and always create a new `orig.tar.gz` file that contains the real `orig.tar.gz` in its toplevel directory. This is questionable with respect to the preference for pristine source. On the other hand, it is easy to modify or add binary files in this case: Just put them into the newly created `orig.tar.gz` file, besides the real one, and copy them to the right place during the build process.

[2] As a special exception, if the omission of non-free files would lead to the source failing to build without assistance from the Debian diff, it might be appropriate to instead edit the files, omitting only the non-free parts of them, and/or explain the situation in a README.Debian-source file in the root of the source tree. But in that case please also urge the upstream author to make the non-free components easier seperable from the rest of the source.

[3] The file should have a name that makes it clear which binary file it encodes. Usually, some postfix indicating the encoding should be appended to the original filename. Note that you don't need to depend on `sharutils` to get the `uudecode` program if you use `perl`'s `pack` function. The code could look like

```
uuencode-file: perl -ne 'print(pack "u", $$_);' $(file) > $(file).uuencoded
uudecode-file: perl -ne 'print(unpack "u", $$_);' $(file).uuencoded > $(file)
```

6.7.9 Best practices for debug packages

A debug package is a package with a name ending in "-dbg", that contains additional information that gdb can use. Since Debian binaries are stripped by default, debugging information, including function names and line numbers, is otherwise not available when running gdb on Debian binaries. Debug packages allow users who need this additional debugging information to install it, without bloating a regular system with the information.

It is up to a package's maintainer whether to create a debug package or not. Maintainers are encouraged to create debug packages for library packages, since this can aid in debugging many programs linked to a library. In general, debug packages do not need to be added for all programs; doing so would bloat the archive. But if a maintainer finds that users often need a debugging version of a program, it can be worthwhile to make a debug package for it. Programs that are core infrastructure, such as apache and the X server are also good candidates for debug packages.

Some debug packages may contain an entire special debugging build of a library or other binary, but most of them can save space and build time by instead containing separated debugging symbols that gdb can find and load on the fly when debugging a program or library. The convention in Debian is to keep these symbols in `/usr/lib/debug/`*`path`*, where *path* is the path to the executable or library. For example, debugging symbols for `/usr/bin/foo` go in `/usr/lib/debug/usr/bin/foo`, and debugging symbols for `/usr/lib/libfoo.so.1` go in `/usr/lib/debug/usr/lib/libfoo.so.1`.

The debugging symbols can be extracted from an object file using "objcopy –only-keep-debug". Then the object file can be stripped, and "objcopy –add-gnu-debuglink" used to specify the path to the debugging symbol file. `objcopy(1)` explains in detail how this works.

The dh_strip command in debhelper supports creating debug packages, and can take care of using objcopy to separate out the debugging symbols for you. If your package uses debhelper, all you need to do is call "dh_strip –dbg-package=libfoo-dbg", and add an entry to debian/control for the debug package.

Note that the Debian package should depend on the package that it provides debugging symbols for, and this dependency should be versioned. For example:

```
Depends: libfoo-dbg (= ${binary:Version})
```

Chapter 7

Beyond Packaging

Debian is about a lot more than just packaging software and maintaining those packages. This chapter contains information about ways, often really critical ways, to contribute to Debian beyond simply creating and maintaining packages.

As a volunteer organization, Debian relies on the discretion of its members in choosing what they want to work on and in choosing the most critical thing to spend their time on.

7.1 Bug reporting

We encourage you to file bugs as you find them in Debian packages. In fact, Debian developers are often the first line testers. Finding and reporting bugs in other developers' packages improves the quality of Debian.

Read the instructions for reporting bugs (`http://www.debian.org/Bugs/Reporting`) in the Debian bug tracking system (`http://www.debian.org/Bugs/`).

Try to submit the bug from a normal user account at which you are likely to receive mail, so that people can reach you if they need further information about the bug. Do not submit bugs as root.

You can use a tool like `reportbug(1)` to submit bugs. It can automate and generally ease the process.

Make sure the bug is not already filed against a package. Each package has a bug list easily reachable at `http://bugs.debian.org/`*`packagename`* Utilities like `querybts(1)` can also provide you with this information (and `reportbug` will usually invoke `querybts` before sending, too).

Try to direct your bugs to the proper location. When for example your bug is about a package which overwrites files from another package, check the bug lists for *both* of those packages in order to avoid filing duplicate bug reports.

For extra credit, you can go through other packages, merging bugs which are reported more than once, or tagging bugs 'fixed' when they have already been fixed. Note that when you are

neither the bug submitter nor the package maintainer, you should not actually close the bug (unless you secure permission from the maintainer).

From time to time you may want to check what has been going on with the bug reports that you submitted. Take this opportunity to close those that you can't reproduce anymore. To find out all the bugs you submitted, you just have to visit `http://bugs.debian.org/from:<your-email-addr>`.

7.1.1 Reporting lots of bugs at once (mass bug filing)

Reporting a great number of bugs for the same problem on a great number of different packages — i.e., more than 10 — is a deprecated practice. Take all possible steps to avoid submitting bulk bugs at all. For instance, if checking for the problem can be automated, add a new check to `lintian` so that an error or warning is emitted.

If you report more than 10 bugs on the same topic at once, it is recommended that you send a message to `<debian-devel@lists.debian.org>` describing your intention before submitting the report, and mentioning the fact in the subject of your mail. This will allow other developers to verify that the bug is a real problem. In addition, it will help prevent a situation in which several maintainers start filing the same bug report simultaneously.

Please use the programms `dd-list` and if appropriate `whodepends` (from the package devscripts) to generate a list of all affected packages, and include the output in your mail to `<debian-devel@lists.debian.org>`.

Note that when sending lots of bugs on the same subject, you should send the bug report to `<maintonly@bugs.debian.org>` so that the bug report is not forwarded to the bug distribution mailing list.

7.2 Quality Assurance effort

7.2.1 Daily work

Even though there is a dedicated group of people for Quality Assurance, QA duties are not reserved solely for them. You can participate in this effort by keeping your packages as bug-free as possible, and as lintian-clean (see '`lintian`' on page 100) as possible. If you do not find that possible, then you should consider orphaning some of your packages (see 'Orphaning a package' on page 47). Alternatively, you may ask the help of other people in order to catch up with the backlog of bugs that you have (you can ask for help on `<debian-qa@lists.debian.org>` or `<debian-devel@lists.debian.org>`). At the same time, you can look for co-maintainers (see 'Collaborative maintenance' on page 58).

7.2.2 Bug squashing parties

From time to time the QA group organizes bug squashing parties to get rid of as many problems as possible. They are announced on `<debian-devel-announce@lists.debian.`

org> and the announcement explains which area will be the focus of the party: usually they focus on release critical bugs but it may happen that they decide to help finish a major upgrade (like a new perl version which requires recompilation of all the binary modules).

The rules for non-maintainer uploads differ during the parties because the announcement of the party is considered prior notice for NMU. If you have packages that may be affected by the party (because they have release critical bugs for example), you should send an update to each of the corresponding bug to explain their current status and what you expect from the party. If you don't want an NMU, or if you're only interested in a patch, or if you will deal yourself with the bug, please explain that in the BTS.

People participating in the party have special rules for NMU, they can NMU without prior notice if they upload their NMU to DELAYED/3-day at least. All other NMU rules apply as usually; they should send the patch of the NMU to the BTS (to one of the open bugs fixed by the NMU, or to a new bug, tagged fixed). They should also respect any particular wishes of the maintainer.

If you don't feel confident about doing an NMU, just send a patch to the BTS. It's far better than a broken NMU.

7.3 Contacting other maintainers

During your lifetime within Debian, you will have to contact other maintainers for various reasons. You may want to discuss a new way of cooperating between a set of related packages, or you may simply remind someone that a new upstream version is available and that you need it.

Looking up the email address of the maintainer for the package can be distracting. Fortunately, there is a simple email alias, `<package>@packages.debian.org`, which provides a way to email the maintainer, whatever their individual email address (or addresses) may be. Replace `<package>` with the name of a source or a binary package.

You may also be interested in contacting the persons who are subscribed to a given source package via 'The Package Tracking System' on page 25. You can do so by using the `<package>@packages.qa.debian.org` email address.

7.4 Dealing with inactive and/or unreachable maintainers

If you notice that a package is lacking maintenance, you should make sure that the maintainer is active and will continue to work on their packages. It is possible that they are not active any more, but haven't registered out of the system, so to speak. On the other hand, it is also possible that they just need a reminder.

There is a simple system (the MIA database) in which information about maintainers who are deemed Missing In Action is recorded. When a member of the QA group contacts an inactive maintainer or finds more information about one, this is recorded in the MIA database. This

system is available in /org/qa.debian.org/mia on the host qa.debian.org, and can be queried with a tool known as `mia-query`. Use

```
mia-query --help
```

to see how to query the database. If you find that no information has been recorded about an inactive maintainer yet, or that you can add more information, you should generally proceed as follows.

The first step is to politely contact the maintainer, and wait a reasonable time for a response. It is quite hard to define "reasonable time", but it is important to take into account that real life is sometimes very hectic. One way to handle this would be to send a reminder after two weeks.

If the maintainer doesn't reply within four weeks (a month), one can assume that a response will probably not happen. If that happens, you should investigate further, and try to gather as much useful information about the maintainer in question as possible. This includes:

- The "echelon" information available through the developers' LDAP database (`https://db.debian.org/`), which indicates when the developer last posted to a Debian mailing list. (This includes uploads via debian-*-changes lists.) Also, remember to check whether the maintainer is marked as "on vacation" in the database.
- The number of packages this maintainer is responsible for, and the condition of those packages. In particular, are there any RC bugs that have been open for ages? Furthermore, how many bugs are there in general? Another important piece of information is whether the packages have been NMUed, and if so, by whom.
- Is there any activity of the maintainer outside of Debian? For example, they might have posted something recently to non-Debian mailing lists or news groups.

A bit of a problem are packages which were sponsored — the maintainer is not an official Debian developer. The echelon information is not available for sponsored people, for example, so you need to find and contact the Debian developer who has actually uploaded the package. Given that they signed the package, they're responsible for the upload anyhow, and are likely to know what happened to the person they sponsored.

It is also allowed to post a query to `<debian-devel@lists.debian.org>`, asking if anyone is aware of the whereabouts of the missing maintainer. Please Cc: the person in question.

Once you have gathered all of this, you can contact `<mia@qa.debian.org>`. People on this alias will use the information you provide in order to decide how to proceed. For example, they might orphan one or all of the packages of the maintainer. If a package has been NMUed, they might prefer to contact the NMUer before orphaning the package — perhaps the person who has done the NMU is interested in the package.

One last word: please remember to be polite. We are all volunteers and cannot dedicate all of our time to Debian. Also, you are not aware of the circumstances of the person who is involved. Perhaps they might be seriously ill or might even have died — you do not know who may be

on the receiving side. Imagine how a relative will feel if they read the e-mail of the deceased and find a very impolite, angry and accusing message!

On the other hand, although we are volunteers, we do have a responsibility. So you can stress the importance of the greater good — if a maintainer does not have the time or interest anymore, they should "let go" and give the package to someone with more time.

If you are interested in working in the MIA team, please have a look at the README file in /org/qa.debian.org/mia on qa.debian.org where the technical details and the MIA procedures are documented and contact <mia@qa.debian.org>.

7.5 Interacting with prospective Debian developers

Debian's success depends on its ability to attract and retain new and talented volunteers. If you are an experienced developer, we recommend that you get involved with the process of bringing in new developers. This section describes how to help new prospective developers.

7.5.1 Sponsoring packages

Sponsoring a package means uploading a package for a maintainer who is not able to do it on their own, a new maintainer applicant. Sponsoring a package also means accepting responsibility for it.

New maintainers usually have certain difficulties creating Debian packages — this is quite understandable. That is why the sponsor is there, to check the package and verify that it is good enough for inclusion in Debian. (Note that if the sponsored package is new, the ftpmasters will also have to inspect it before letting it in.)

Sponsoring merely by signing the upload or just recompiling is **definitely not recommended**. You need to build the source package just like you would build a package of your own. Remember that it doesn't matter that you left the prospective developer's name both in the changelog and the control file, the upload can still be traced to you.

If you are an application manager for a prospective developer, you can also be their sponsor. That way you can also verify how the applicant is handling the 'Tasks and Skills' part of their application.

7.5.2 Managing sponsored packages

By uploading a sponsored package to Debian, you are certifying that the package meets minimum Debian standards. That implies that you must build and test the package on your own system before uploading.

You cannot simply upload a binary `.deb` from the sponsoree. In theory, you should only ask for the diff file and the location of the original source tarball, and then you should download the source and apply the diff yourself. In practice, you may want to use the source package

built by your sponsoree. In that case, you have to check that they haven't altered the upstream files in the `.orig.tar.gz` file that they're providing.

Do not be afraid to write the sponsoree back and point out changes that need to be made. It often takes several rounds of back-and-forth email before the package is in acceptable shape. Being a sponsor means being a mentor.

Once the package meets Debian standards, build and sign it with

```
dpkg-buildpackage -kKEY-ID
```

before uploading it to the incoming directory. Of course, you can also use any part of your *KEY-ID*, as long as it's unique in your secret keyring.

The Maintainer field of the `control` file and the `changelog` should list the person who did the packaging, i.e., the sponsoree. The sponsoree will therefore get all the BTS mail about the package.

If you prefer to leave a more evident trace of your sponsorship job, you can add a line stating it in the most recent changelog entry.

You are encouraged to keep tabs on the package you sponsor using 'The Package Tracking System' on page 25.

7.5.3 Advocating new developers

See the page about advocating a prospective developer (`http://www.debian.org/devel/join/nm-advocate`) at the Debian web site.

7.5.4 Handling new maintainer applications

Please see Checklist for Application Managers (`http://www.debian.org/devel/join/nm-amchecklist`) at the Debian web site.

Chapter 8

Internationalizing, translating, being internationalized and being translated

Debian supports an ever-increasing number of natural languages. Even if you are a native English speaker and do not speak any other language, it is part of your duty as a maintainer to be aware of issues of internationalization (abbreviated i18n because there are 18 letters between the 'i' and the 'n' in internationalization). Therefore, even if you are ok with English-only programs, you should read most of this chapter.

According to Introduction to i18n (http://www.debian.org/doc/manuals/intro-i18n/) from Tomohiro KUBOTA, "I18N (internationalization) means modification of a software or related technologies so that a software can potentially handle multiple languages, customs, and so on in the world." while "L10N (localization) means implementation of a specific language for an already internationalized software."

l10n and i18n are interconnected, but the difficulties related to each of them are very different. It's not really difficult to allow a program to change the language in which texts are displayed based on user settings, but it is very time consuming to actually translate these messages. On the other hand, setting the character encoding is trivial, but adapting the code to use several character encodings is a really hard problem.

Setting aside the i18n problems, where no general guideline can be given, there is actually no central infrastructure for l10n within Debian which could be compared to the dbuild mechanism for porting. So most of the work has to be done manually.

8.1 How translations are handled within Debian

Handling translation of the texts contained in a package is still a manual task, and the process depends on the kind of text you want to see translated.

For program messages, the gettext infrastructure is used most of the time. Most of the time, the translation is handled upstream within projects like the Free Translation Project (http://www.iro.umontreal.ca/contrib/po/HTML/), the Gnome translation Project (http:

//developer.gnome.org/projects/gtp/) or the KDE one (http://i18n.kde.org/). The only centralized resource within Debian is the Central Debian translation statistics (http://www.debian.org/intl/l10n/), where you can find some statistics about the translation files found in the actual packages, but no real infrastructure to ease the translation process.

An effort to translate the package descriptions started long ago, even if very little support is offered by the tools to actually use them (i.e., only APT can use them, when configured correctly). Maintainers don't need to do anything special to support translated package descriptions; translators should use the DDTP (http://ddtp.debian.org/).

For debconf templates, maintainers should use the po-debconf package to ease the work of translators, who could use the DDTP to do their work (but the French and Brazilian teams don't). Some statistics can be found both on the DDTP site (about what is actually translated), and on the Central Debian translation statistics (http://www.debian.org/intl/l10n/) site (about what is integrated in the packages).

For web pages, each l10n team has access to the relevant CVS, and the statistics are available from the Central Debian translation statistics site.

For general documentation about Debian, the process is more or less the same as for the web pages (the translators have access to the CVS), but there are no statistics pages.

For package-specific documentation (man pages, info documents, other formats), almost everything remains to be done.

Most notably, the KDE project handles translation of its documentation in the same way as its program messages.

There is an effort to handle Debian-specific man pages within a specific CVS repository (http://cvs.debian.org/manpages/?cvsroot=debian-doc).

8.2 I18N & L10N FAQ for maintainers

This is a list of problems that maintainers may face concerning i18n and l10n. While reading this, keep in mind that there is no real consensus on these points within Debian, and that this is only advice. If you have a better idea for a given problem, or if you disagree on some points, feel free to provide your feedback, so that this document can be enhanced.

8.2.1 How to get a given text translated

To translate package descriptions or debconf templates, you have nothing to do; the DDTP infrastructure will dispatch the material to translate to volunteers with no need for interaction from your part.

For all other material (gettext files, man pages, or other documentation), the best solution is to put your text somewhere on the Internet, and ask on debian-i18n for a translation in different languages. Some translation team members are subscribed to this list, and they will take care of the translation and of the reviewing process. Once they are done, you will get your translated document from them in your mailbox.

8.2.2 How to get a given translation reviewed

From time to time, individuals translate some texts in your package and will ask you for inclusion of the translation in the package. This can become problematic if you are not fluent in the given language. It is a good idea to send the document to the corresponding l10n mailing list, asking for a review. Once it has been done, you should feel more confident in the quality of the translation, and feel safe to include it in your package.

8.2.3 How to get a given translation updated

If you have some translations of a given text lying around, each time you update the original, you should ask the previous translator to update the translation with your new changes. Keep in mind that this task takes time; at least one week to get the update reviewed and all.

If the translator is unresponsive, you may ask for help on the corresponding l10n mailing list. If everything fails, don't forget to put a warning in the translated document, stating that the translation is somehow outdated, and that the reader should refer to the original document if possible.

Avoid removing a translation completely because it is outdated. Old documentation is often better than no documentation at all for non-English speakers.

8.2.4 How to handle a bug report concerning a translation

The best solution may be to mark the bug as "forwarded to upstream", and forward it to both the previous translator and his/her team (using the corresponding debian-l10n-XXX mailing list).

8.3 I18N & L10N FAQ for translators

While reading this, please keep in mind that there is no general procedure within Debian concerning these points, and that in any case, you should collaborate with your team and the package maintainer.

8.3.1 How to help the translation effort

Choose what you want to translate, make sure that nobody is already working on it (using your debian-l10n-XXX mailing list), translate it, get it reviewed by other native speakers on your l10n mailing list, and provide it to the maintainer of the package (see next point).

8.3.2 How to provide a translation for inclusion in a package

Make sure your translation is correct (asking for review on your l10n mailing list) before providing it for inclusion. It will save time for everyone, and avoid the chaos resulting in having several versions of the same document in bug reports.

The best solution is to file a regular bug containing the translation against the package. Make sure to use the 'PATCH' tag, and to not use a severity higher than 'wishlist', since the lack of translation never prevented a program from running.

8.4 Best current practice concerning l10n

- As a maintainer, never edit the translations in any way (even to reformat the layout) without asking on the corresponding l10n mailing list. You risk for example breaksing the encoding of the file by doing so. Moreover, what you consider an error can be right (or even needed) in the given language.
- As a translator, if you find an error in the original text, make sure to report it. Translators are often the most attentive readers of a given text, and if they don't report the errors they find, nobody will.
- In any case, remember that the major issue with l10n is that it requires several people to cooperate, and that it is very easy to start a flamewar about small problems because of misunderstandings. So if you have problems with your interlocutor, ask for help on the corresponding l10n mailing list, on debian-i18n, or even on debian-devel (but beware, l10n discussions very often become flamewars on that list :)
- In any case, cooperation can only be achieved with **mutual respect**.

Appendix A

Overview of Debian Maintainer Tools

This section contains a rough overview of the tools available to maintainers. The following is by no means complete or definitive, but just a guide to some of the more popular tools.

Debian maintainer tools are meant to aid developers and free their time for critical tasks. As Larry Wall says, there's more than one way to do it.

Some people prefer to use high-level package maintenance tools and some do not. Debian is officially agnostic on this issue; any tool which gets the job done is fine. Therefore, this section is not meant to stipulate to anyone which tools they should use or how they should go about their duties of maintainership. Nor is it meant to endorse any particular tool to the exclusion of a competing tool.

Most of the descriptions of these packages come from the actual package descriptions themselves. Further information can be found in the package documentation itself. You can also see more info with the command `apt-cache show <package-name>`.

A.1 Core tools

The following tools are pretty much required for any maintainer.

A.1.1 `dpkg-dev`

`dpkg-dev` contains the tools (including `dpkg-source`) required to unpack, build, and upload Debian source packages. These utilities contain the fundamental, low-level functionality required to create and manipulate packages; as such, they are essential for any Debian maintainer.

A.1.2 `debconf`

`debconf` provides a consistent interface to configuring packages interactively. It is user interface independent, allowing end-users to configure packages with a text-only interface, an

HTML interface, or a dialog interface. New interfaces can be added as modules.

You can find documentation for this package in the `debconf-doc` package.

Many feel that this system should be used for all packages which require interactive configuration; see 'Configuration management with `debconf`' on page 75. `debconf` is not currently required by Debian Policy, but that may change in the future.

A.1.3 `fakeroot`

`fakeroot` simulates root privileges. This enables you to build packages without being root (packages usually want to install files with root ownership). If you have `fakeroot` installed, you can build packages as a regular user: `dpkg-buildpackage -rfakeroot`.

A.2 Package lint tools

According to the Free On-line Dictionary of Computing (FOLDOC), 'lint' is "a Unix C language processor which carries out more thorough checks on the code than is usual with C compilers." Package lint tools help package maintainers by automatically finding common problems and policy violations in their packages.

A.2.1 `lintian`

`lintian` dissects Debian packages and emits information about bugs and policy violations. It contains automated checks for many aspects of Debian policy as well as some checks for common errors.

You should periodically get the newest `lintian` from 'unstable' and check over all your packages. Notice that the `-i` option provides detailed explanations of what each error or warning means, what its basis in Policy is, and commonly how you can fix the problem.

Refer to 'Testing the package' on page 32 for more information on how and when to use Lintian.

You can also see a summary of all problems reported by Lintian on your packages at `http://lintian.debian.org/`. These reports contain the latest `lintian` output for the whole development distribution ("unstable").

A.2.2 `linda`

`linda` is another package linter. It is similar to `lintian` but has a different set of checks. Its written in Python rather than Perl.

A.2.3 `debdiff`

`debdiff` (from the `devscripts` package, '`devscripts`' on page 104) compares file lists and control files of two packages. It is a simple regression test, as it will help you notice if the number of binary packages has changed since the last upload, or if something has changed in the control file. Of course, some of the changes it reports will be all right, but it can help you prevent various accidents.

You can run it over a pair of binary packages:

```
debdiff package_1-1_arch.deb package_2-1_arch.deb
```

Or even a pair of changes files:

```
debdiff package_1-1_arch.changes package_2-1_arch.changes
```

For more information please see `debdiff(1)`.

A.3 Helpers for `debian/rules`

Package building tools make the process of writing `debian/rules` files easier. See 'Helper scripts' on page 65 for more information about why these might or might not be desired.

A.3.1 `debhelper`

`debhelper` is a collection of programs which can be used in `debian/rules` to automate common tasks related to building binary Debian packages. `debhelper` includes programs to install various files into your package, compress files, fix file permissions, and integrate your package with the Debian menu system.

Unlike some approaches, `debhelper` is broken into several small, simple commands which act in a consistent manner. As such, it allows more fine-grained control than some of the other "debian/rules tools".

There are a number of little `debhelper` add-on packages, too transient to document. You can see the list of most of them by doing `apt-cache search ^dh-`.

A.3.2 `debmake`

`debmake`, a precursor to `debhelper`, is a more coarse-grained `debian/rules` assistant. It includes two main programs: `deb-make`, which can be used to help a maintainer convert a regular (non-Debian) source archive into a Debian source package; and `debstd`, which incorporates in one big shot the same sort of automated functions that one finds in `debhelper`.

The consensus is that `debmake` is now deprecated in favor of `debhelper`. It is a bug to use `debmake` in new packages. New packages using `debmake` will be rejected from the archive.

A.3.3 dh-make

The dh-make package contains dh_make, a program that creates a skeleton of files necessary to build a Debian package out of a source tree. As the name suggests, dh_make is a rewrite of debmake and its template files use dh_* programs from debhelper.

While the rules files generated by dh_make are in general a sufficient basis for a working package, they are still just the groundwork: the burden still lies on the maintainer to finely tune the generated files and make the package entirely functional and Policy-compliant.

A.3.4 yada

yada is another packaging helper tool. It uses a debian/packages file to auto-generate debian/rules and other necessary files in the debian/ subdirectory. The debian/packages file contains instruction to build packages and there is no need to create any Makefile files. There is possibility to use macro engine similar to the one used in SPECS files from RPM source packages.

For more informations see YADA site (http://yada.alioth.debian.org/).

A.3.5 equivs

equivs is another package for making packages. It is often suggested for local use if you need to make a package simply to fulfill dependencies. It is also sometimes used when making "meta-packages", which are packages whose only purpose is to depend on other packages.

A.4 Package builders

The following packages help with the package building process, general driving dpkg-buildpackage as well as handling supporting tasks.

A.4.1 cvs-buildpackage

cvs-buildpackage provides the capability to inject or import Debian source packages into a CVS repository, build a Debian package from the CVS repository, and helps in integrating upstream changes into the repository.

These utilities provide an infrastructure to facilitate the use of CVS by Debian maintainers. This allows one to keep separate CVS branches of a package for *stable*, *unstable* and possibly *experimental* distributions, along with the other benefits of a version control system.

A.4.2 debootstrap

The debootstrap package and script allows you to "bootstrap" a Debian base system into any part of your filesystem. By "base system", we mean the bare minimum of packages required to operate and install the rest of the system.

Having a system like this can be useful in many ways. For instance, you can chroot into it if you want to test your build dependencies. Or you can test how your package behaves when installed into a bare base system. Chroot builders use this package; see below.

A.4.3 pbuilder

pbuilder constructs a chrooted system, and builds a package inside the chroot. It is very useful to check that a package's build-dependencies are correct, and to be sure that unnecessary and wrong build dependencies will not exist in the resulting package.

A related package is pbuilder-uml, which goes even further by doing the build within a User Mode Linux environment.

A.4.4 sbuild

sbuild is another automated builder. It can use chrooted environments as well. It can be used stand-alone, or as part of a networked, distributed build environment. As the latter, it is part of the system used by porters to build binary packages for all the available architectures. See 'buildd' on page 52 for more information, and http://buildd.debian.org/ to see the system in action.

A.5 Package uploaders

The following packages help automate or simplify the process of uploading packages into the official archive.

A.5.1 dupload

dupload is a package and a script to automatically upload Debian packages to the Debian archive, to log the upload, and to send mail about the upload of a package. You can configure it for new upload locations or methods.

A.5.2 dput

The dput package and script does much the same thing as dupload, but in a different way. It has some features over dupload, such as the ability to check the GnuPG signature and

checksums before uploading, and the possibility of running `dinstall` in dry-run mode after the upload.

A.5.3 `dcut`

The `dcut` script (part of the package '`dput`' on the preceding page) helps in removing files from the ftp upload directory.

A.6 Maintenance automation

The following tools help automate different maintenance tasks, from adding changelog entries or signature lines and looking up bugs in Emacs to making use of the newest and official `config.sub`.

A.6.1 `devscripts`

`devscripts` is a package containing wrappers and tools which are very helpful for maintaining your Debian packages. Example scripts include `debchange` and `dch`, which manipulate your `debian/changelog` file from the command-line, and `debuild`, which is a wrapper around `dpkg-buildpackage`. The `bts` utility is also very helpful to update the state of bug reports on the command line. `uscan` can be used to watch for new upstream versions of your packages. `debrsign` can be used to remotely sign a package prior to upload, which is nice when the machine you build the package on is different from where your GPG keys are.

See the `devscripts(1)` manual page for a complete list of available scripts.

A.6.2 `autotools-dev`

`autotools-dev` contains best practices for people who maintain packages which use `autoconf` and/or `automake`. Also contains canonical `config.sub` and `config.guess` files which are known to work on all Debian ports.

A.6.3 `dpkg-repack`

`dpkg-repack` creates Debian package file out of a package that has already been installed. If any changes have been made to the package while it was unpacked (e.g., files in `/etc` were modified), the new package will inherit the changes.

This utility can make it easy to copy packages from one computer to another, or to recreate packages which are installed on your system but no longer available elsewhere, or to save the current state of a package before you upgrade it.

A.6.4 alien

`alien` converts binary packages between various packaging formats, including Debian, RPM (RedHat), LSB (Linux Standard Base), Solaris, and Slackware packages.

A.6.5 debsums

`debsums` checks installed packages against their MD5 sums. Note that not all packages have MD5 sums, since they aren't required by Policy.

A.6.6 dpkg-dev-el

`dpkg-dev-el` is an Emacs lisp package which provides assistance when editing some of the files in the `debian` directory of your package. For instance, there are handy functions for listing a package's current bugs, and for finalizing the latest entry in a `debian/changelog` file.

A.6.7 dpkg-depcheck

`dpkg-depcheck` (from the `devscripts` package, '`devscripts`' on the preceding page) runs a command under `strace` to determine all the packages that were used by the said command.

For Debian packages, this is useful when you have to compose a `Build-Depends` line for your new package: running the build process through `dpkg-depcheck` will provide you with a good first approximation of the build-dependencies. For example:

```
dpkg-depcheck -b debian/rules build
```

`dpkg-depcheck` can also be used to check for run-time dependencies, especially if your package uses exec(2) to run other programs.

For more information please see `dpkg-depcheck(1)`.

A.7 Porting tools

The following tools are helpful for porters and for cross-compilation.

A.7.1 quinn-diff

`quinn-diff` is used to locate the differences from one architecture to another. For instance, it could tell you which packages need to be ported for architecture *Y*, based on architecture *X*.

A.7.2 `dpkg-cross`

`dpkg-cross` is a tool for installing libraries and headers for cross-compiling in a way similar to `dpkg`. Furthermore, the functionality of `dpkg-buildpackage` and `dpkg-shlibdeps` is enhanced to support cross-compiling.

A.8 Documentation and information

The following packages provide information for maintainers or help with building documentation.

A.8.1 `debiandoc-sgml`

`debiandoc-sgml` provides the DebianDoc SGML DTD, which is commonly used for Debian documentation. This manual, for instance, is written in DebianDoc. It also provides scripts for building and styling the source to various output formats.

Documentation for the DTD can be found in the `debiandoc-sgml-doc` package.

A.8.2 `debian-keyring`

Contains the public GPG and PGP keys of Debian developers. See 'Maintaining your public key' on page 7 and the package documentation for more information.

A.8.3 `debview`

`debview` provides an Emacs mode for viewing Debian binary packages. This lets you examine a package without unpacking it.