

Debian Developer's Reference

Developer's Reference Team <developers-reference@packages.debian.org>

Adam Di Carlo, editor

Raphaël Hertzog

Christian Schwarz

Ian Jackson

ver. 3.3.4, 29 February, 2004

Copyright Notice

copyright © 1998—2003 Adam Di Carlo
copyright © 2002—2003 Raphaël Hertzog
copyright © 1997, 1998 Christian Schwarz

This manual is free software; you may redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version.

This is distributed in the hope that it will be useful, but *without any warranty*; without even the implied warranty of merchantability or fitness for a particular purpose. See the GNU General Public License for more details.

A copy of the GNU General Public License is available as `/usr/share/common-licenses/GPL` in the Debian GNU/Linux distribution or on the World Wide Web at the GNU web site (<http://www.gnu.org/copyleft/gpl.html>). You can also obtain it by writing to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Contents

1	Scope of This Document	1
2	Applying to Become a Maintainer	3
2.1	Getting started	3
2.2	Registering as a Debian developer	4
2.3	Debian mentors and sponsors	5
3	Debian Developer's Duties	7
3.1	Maintaining your Debian information	7
3.2	Maintaining your public key	7
3.3	Voting	8
3.4	Going on vacation gracefully	8
3.5	Coordination with upstream developers	9
3.6	Managing release-critical bugs	9
3.7	Retiring	9
4	Resources for Debian Developers	11
4.1	Mailing lists	11
4.1.1	Basic rules for use	11
4.1.2	Core development mailing lists	12
4.1.3	Special lists	12
4.1.4	Requesting new development-related lists	12

4.2	IRC channels	13
4.3	Documentation	13
4.4	Debian machines	14
4.4.1	The bugs server	14
4.4.2	The ftp-master server	15
4.4.3	The non-US server	15
4.4.4	The www-master server	15
4.4.5	The people web server	15
4.4.6	The CVS server	16
4.5	The Developers Database	16
4.6	The Debian archive	17
4.6.1	Sections	19
4.6.2	Architectures	20
4.6.3	Packages	20
4.6.4	Distributions	20
4.6.5	Release code names	23
4.7	Debian mirrors	24
4.8	The Incoming system	24
4.8.1	Delayed incoming	25
4.9	Package information	26
4.9.1	On the web	26
4.9.2	The madison utility	26
4.10	The Package Tracking System	27
4.10.1	The PTS email interface	28
4.10.2	Filtering PTS mails	29
4.10.3	Forwarding CVS commits in the PTS	29
4.10.4	The PTS web interface	29
4.11	Developer's packages overview	31
4.12	Debian *Forge: Alioth	31

5	Managing Packages	33
5.1	New packages	33
5.2	Recording changes in the package	34
5.3	Testing the package	34
5.4	Layout of the source package	35
5.5	Picking a distribution	36
5.5.1	Special case: uploads to the <i>stable</i> distribution	36
5.5.2	Special case: uploads to <i>testing-proposed-updates</i>	37
5.6	Uploading a package	37
5.6.1	Uploading to <i>ftp-master</i>	37
5.6.2	Uploading to non-US	38
5.6.3	Uploads via <i>chiark</i>	39
5.6.4	Uploads via <i>erlangen</i>	39
5.6.5	Other upload queues	40
5.6.6	Notification that a new package has been installed	40
5.7	Determining section and priority of a package	40
5.8	Handling bugs	41
5.8.1	Monitoring bugs	41
5.8.2	Responding to bugs	42
5.8.3	Bug housekeeping	42
5.8.4	When bugs are closed by new uploads	44
5.8.5	Handling security-related bugs	45
5.9	Moving, removing, renaming, adopting, and orphaning packages	49
5.9.1	Moving packages	49
5.9.2	Removing packages	49
5.9.3	Replacing or renaming packages	50
5.9.4	Orphaning a package	51
5.9.5	Adopting a package	51
5.10	Porting and being ported	52

5.10.1	Being kind to porters	52
5.10.2	Guidelines for porter uploads	53
5.10.3	Porting infrastructure and automation	55
5.11	Non-Maintainer Uploads (NMUs)	56
5.11.1	Terminology	56
5.11.2	Who can do an NMU	57
5.11.3	When to do a source NMU	57
5.11.4	How to do a source NMU	58
5.11.5	Acknowledging an NMU	60
5.12	Collaborative maintenance	61
6	Best Packaging Practices	63
6.1	Best practices for <code>debian/rules</code>	63
6.1.1	Helper scripts	63
6.1.2	Separating your patches into multiple files	64
6.1.3	Multiple binary packages	65
6.2	Best practices for <code>debian/control</code>	65
6.2.1	General guidelines for package descriptions	65
6.2.2	The package synopsis, or short description	66
6.2.3	The long description	67
6.2.4	Upstream home page	67
6.3	Best practices for <code>debian/changelog</code>	68
6.3.1	Writing useful changelog entries	68
6.3.2	Common misconceptions about changelog entries	68
6.3.3	Common errors in changelog entries	69
6.4	Best practices for maintainer scripts	70
6.5	Configuration management with <code>debconf</code>	71
6.6	Internationalization	72
6.6.1	Handling <code>debconf</code> translations	72

6.6.2	Internationalized documentation	72
6.7	Common packaging situations	73
6.7.1	Packages using autoconf/automake	73
6.7.2	Libraries	73
6.7.3	Documentation	73
6.7.4	Specific types of packages	73
6.7.5	Architecture-independent data	74
7	Beyond Packaging	75
7.1	Bug reporting	75
7.1.1	Reporting lots of bugs at once	76
7.2	Quality Assurance effort	76
7.2.1	Daily work	76
7.2.2	Bug squashing parties	77
7.3	Contacting other maintainers	77
7.4	Dealing with inactive and/or unreachable maintainers	78
7.5	Interacting with prospective Debian developers	79
7.5.1	Sponsoring packages	79
7.5.2	Managing sponsored packages	79
7.5.3	Advocating new developers	80
7.5.4	Handling new maintainer applications	80
A	Overview of Debian Maintainer Tools	81
A.1	Core tools	81
A.1.1	dpkg-dev	81
A.1.2	debconf	82
A.1.3	fakeroot	82
A.2	Package lint tools	82
A.2.1	lintian	82

A.2.2	linda	83
A.2.3	debdiff	83
A.3	Helpers for debian/rules	83
A.3.1	debhelper	83
A.3.2	debmake	84
A.3.3	dh-make	84
A.3.4	yada	84
A.3.5	equivs	84
A.4	Package builders	84
A.4.1	cvs-buildpackage	85
A.4.2	debootstrap	85
A.4.3	pbuilder	85
A.4.4	sbuid	85
A.5	Package uploaders	85
A.5.1	dupload	86
A.5.2	dput	86
A.6	Maintenance automation	86
A.6.1	devscripts	86
A.6.2	autotools-dev	86
A.6.3	dpkg-repack	87
A.6.4	alien	87
A.6.5	debsums	87
A.6.6	dpkg-dev-el	87
A.6.7	dpkg-depcheck	87
A.7	Porting tools	88
A.7.1	quinn-diff	88
A.7.2	dpkg-cross	88
A.8	Documentation and information	88
A.8.1	debiandoc-sgml	88

A.8.2	debian-keyring	88
A.8.3	debview	88

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 33) and how to upload packages ('Uploading a package' on page 37); how to handle bug reports ('Handling bugs' on page 41); how to move, remove, or orphan packages ('Moving, removing, renaming, adopting, and orphaning packages' on page 49); how to port packages ('Porting and being ported' on page 52); and how and when to do interim releases of other maintainers' packages ('Non-Maintainer Uploads (NMUs)' on page 56).

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 37); and a discussion of resources which can help maintainers with the quality of their packages ('Overview of Debian Maintainer Tools' on page 81).

It should be clear that this reference does not discuss the technical details of the Debian package nor how to generate Debian packages. 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 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 page 5 for details. The IRC channel `#debian` can also be helpful.

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 79 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.

2.2 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 exactly 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 800 people 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 can show this by submitting patches through the Bug Tracking System or having a package sponsored by an existing maintainer 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 maintainer 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 maintainer close to you (If you cannot find a Debian maintainer close to you, there's an alternative way to pass the ID check. You can send in a photo ID signed with your GnuPG key. Having your GnuPG key signed is the preferred way, however. See the identification page (<http://www.debian.org/devel/join/nm-step2>) for more information about these two options.)

If you do not have an OpenPGP key yet, generate one. Every developer needs a 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.gnupg.org/rfc2440.html>).

The recommended public key algorithm for use in Debian development work is DSA (sometimes call “DSS” or “DH/ElGamal”). Other key types may be used however. 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. Your key must be signed with at least your own user ID; this prevents user ID tampering. `gpg` does this automatically.

If your public key isn’t on public key servers such as `pgp5.ai.mit.edu`, please read the documentation available locally in `/usr/share/doc/pgp/keyserv.doc`. 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. Debian GNU/Linux does not require the use of cryptography *qua* cryptography in any manner. If you live in 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 maintainer to verify 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.

2.3 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 maintainers, and who are willing to criticize and upload your packages for you. Those who are seeking a sponsor can request one at <http://www.internatif.org/bortzmeyer/debian/sponsor/>.

If you wish to be a mentor and/or sponsor, more information is available in 'Interacting with prospective Debian developers' on page 79.

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/>).

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, send mail to `<keyring-maint@debian.org>`. 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 emails messages.

The list of all the 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 the 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 56) your package if a big problem (release critical bugs, security update, etc.) occurs while you're on vacation. Sometimes it's nothing as critical as that, but it's still appropriate to let the others know that you're unavailable.

In order to inform the other developers, there's two things that you should do. First send a mail to <debian-private@lists.debian.org> with "[VAC]" prepended to the subject of your message¹ 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!

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

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 41. 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. Those 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 56) 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 51.

- 2 Send an 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 emailing to <keyring-maint@debian.org>.

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.

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 freenode (<http://www.freenode.net/>) network (previously known as Open Projects Network). The `irc.debian.org` DNS entry is an alias to `irc.freenode.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` it's 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 squash parties. `#debian-boot` is used to coordinate the work on the boot floppies (i.e., the 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-sf` (SourceForge package), `#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 Open and free technology community (OFTC) (<http://www.oftc.net/>) IRC network.

4.3 Documentation

This document contains a lot of information very useful to Debian developers, but it can not 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.

Most 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, the Debian system administrator team is reachable at debian-admin@lists.debian.org.

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 75 for information on how to submit bugs.

4.4.1 The bugs server

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

All Debian developers have accounts on `bugs.debian.org`.

4.4.2 The ftp-master server

The `ftp-master.debian.org` server holds the canonical copy of the Debian archive (excluding the non-US packages). Generally, package uploads go to this server; see ‘Uploading a package’ on page 37.

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 49.

4.4.3 The non-US server

The non-US server, `non-us.debian.org`, holds the canonical copy of the non-US part of the Debian archive. If you need to upload a package into one of the non-US sections, upload it to this server; see ‘Uploading to non-US’ on page 38.

Problems with the non-US package archive should generally be submitted as bugs against the `nonus.debian.org` pseudo-package (notice the lack of hyphen between “non” and “us” in the pseudo-package name — that’s for backwards compatibility). Remember to check whether or not someone else has already reported the problem on the Bug Tracking System (<http://bugs.debian.org/nonus.debian.org>).

4.4.4 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 on the Bug Tracking System (<http://bugs.debian.org/www.debian.org>).

4.4.5 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, such as the aforementioned `non-us.debian.org`.

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

4.4.6 The CVS server

Our CVS server is located on `cvs.debian.org`.

If you need to use a publicly accessible CVS server, for instance, to help coordinate work on a package between many different developers, you can request a CVS area on the server.

Generally, `cvs.debian.org` offers a combination of local CVS access, anonymous client-server read-only access, and full client-server access through `ssh`. Also, the CVS area can be accessed read-only via the Web at <http://cvs.debian.org/>.

To request a CVS area, send a request via email to `<debian-admin@debian.org>`. Include the name of the requested CVS area, the Debian account that should own the CVS root area, and why you need it.

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>.

One 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 he’s allowed to. (Since this varies greatly from vendor to vendor, this job can’t be done by the Debian developers.)

Note also 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 the Intel i386 (or greater) platforms, and so was Debian. But when 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 adds 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 from 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 is comprised of 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`), one called *testing* (residing in `dists/testing`), and one called *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 changes from day-to-day. Since no special effort is done to make sure everything in this distribution is working properly, it is sometimes literally unstable.

"testing" 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 'More information about the testing distribution' on the next page.

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 goes into a 'deep freeze', when no changes are made to it except those needed for the installation system. This is called a "test cycle", and it can last up to two weeks. There can be several test cycles, until the distribution is prepared for release, as decided by the release manager. At the end of the last test cycle, the *testing* distribution is renamed to *stable*, overriding the old *stable* distribution, which is removed at that time (although it can be found at 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).

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

The scripts that update the *testing* distribution are run each day after the installation of the updated packages. They generate the `packages` files for the *testing* distribution, but they do so in an intelligent manner trying to avoid any inconsistency and trying 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 several days; the precise number depends on the upload's urgency field. It is 10 days for low urgency, 5 days for medium urgency and 2 days for high urgency. Those delays may be doubled during a freeze;
- It must have less release-critical bugs than the version available in *testing*;
- It must be available on all architectures on which it has been previously built. 'The madison utility' on page 26 may be of interest to check that information;
- It must not break any dependency of a package that 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 if they respect 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 one informed of the progression of their packages into *testing*.

The `update_excuses` file does not always give the precise reason why the package is refused, one may have to find it on their 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. In that case, the release manager must be contacted, and he will force the inclusion of the packages.

In general, please refer to the testing web page (<http://www.debian.org/devel/testing>) for more information. It also includes answers to some of the frequently asked questions.

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

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’; and Debian 3.0, ‘woody’. 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 both on `ftp-master.debian.org` and `non-us.debian.org`.

Packages are uploaded by all the maintainers into a directory called `unchecked`. This directory 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, it is moved in the `new` directory, where it waits for an approval of the `ftpmasters`. If the package contains files to be installed “by-hand” it is moved in the `byhand` directory, where it waits for a manual installation by the `ftpmasters`. Otherwise, if any error has been detected, the package is refused and is moved in the `reject` directory.

Once the package is accepted the system sends a confirmation mail to the maintainer, 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/> (there is no such URL for packages in the non-US archive) until it is really installed in the Debian archive. This happens only once a day, 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.

All Debian developers have write access to the `unchecked` directory in order to upload their packages, they also have that access to the `reject` directory in order to remove their bad uploads or to move some files back in the `unchecked` directory. But all the other directories are only writable by the `ftpmasters`, that is why you can not remove an upload once it has been accepted.

4.8.1 Delayed incoming

The `unchecked` directory has a special `DELAYED` subdirectory. It is itself subdivided in nine directories called `1-day` to `9-day`. Packages which are uploaded in one of those directories will be moved in the real `unchecked` directory after the corresponding number of days. This is done by a script that is run each day and which moves the packages between the directories. Those which are in “1-day” are installed in `unchecked` while the others are moved in the adjacent directory (for example, a package in `5-day` will be moved in `4-day`). This feature is particularly useful for people who are doing non-maintainer uploads. Instead of waiting before uploading a NMU, it is uploaded as soon as it is ready but in one of those `DELAYED/x-day` directories. That leaves the corresponding number of days for the maintainer to react and upload another fix themselves if they are not completely satisfied with the NMU. Alternatively they can remove the NMU.

The use of that delayed feature can be simplified with a bit of integration with your upload tool.

For instance, if you use `dupload` (see ‘dupload’ on page 86), you can add this snippet to your configuration file:

```
$delay = ($ENV{DELAY} || 7);
$cfg{'delayed'} = {
    fqdn => "ftp-master.debian.org",
    login => "yourdebianlogin",
    incoming => "/org/ftp.debian.org/incoming/DELAYED/$delay-day/",
    dinstall_runs => 1,
    method => "scpb"
};
```

Once you’ve made that change, `dupload` can be used to easily upload a package in one of the delayed directories:

```
DELAY=5 dupload --to delayed <changes-file>
```

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 both `ftp-master.debian.org` and `non-us.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, m68k,
libdbd-mysql-perl | 1.2216-2 | testing | source, arm, hppa, i386, ia64,
libdbd-mysql-perl | 1.2216-2.0.1 | testing | alpha
libdbd-mysql-perl | 1.2219-1 | unstable | source, alpha, arm, hppa, i386,
```

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 time 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.

summary (This is a planned expansion.) The regular summary emails about the package's status (bug statistics, porting overview, progression in *testing*, ...).

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 CVS commit notifications, if the package has a CVS repository and the maintainer has set up forwarding commit notifications to the PTS.

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

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.

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
- **cvs**: notification of CVS commits
- **ddtp**: translations of descriptions and debconf templates
- **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.

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.

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
X-Unsubscribe: echo 'unsubscribe dpkg' | mail pts@qa.debian.org
```

4.10.3 Forwarding CVS commits in the PTS

If you use a publicly accessible CVS 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 CVS 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.

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, build 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 bug, and so that you don't forget which packages are under your responsibility.

4.12 Debian *Forge: Alioth

Alioth is a fairly new 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.

For more information please visit <http://alioth.debian.org/>.

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 75) 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 on the changelog of the new package in order for the bug report to be automatically closed once the new package is installed on the archive ('When bugs are closed by new uploads' on page 44).

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.

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 36. 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 44.

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 86 and ‘`dpkg-dev-el`’ on page 87.

See also ‘Best practices for `debian/changelog`’ on page 68.

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 82.

- Optionally run 'debdiff' on page 83 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.

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.

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 45 for more information on those.

It is technically possible to upload a package into several distributions at the same time but it usually doesn't make sense to use that feature because the dependencies of the package may vary with the distribution. In particular, it never makes sense to combine the *experimental* distribution with anything else (see 'Experimental' on page 23).

5.5.1 Special case: uploads to the *stable* distribution

Uploading to *stable* means that the package will be placed into the `stable-proposed-updates` directory of the Debian archive for further testing before it is actually included in *stable*.

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 45 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 in *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.

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

The testing distribution is fed with packages from *unstable* according to the rules explained in 'More information about the testing distribution' on page 22. However, the release manager may stop the testing scripts when he wants to freeze the distribution. In that case, you may want to upload to *testing-proposed-updates* to provide fixed packages during the freeze.

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 manager's eyes, you should read the instructions that he regularly gives 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 it but it is recommended to ask the authorization of the release manager before.

5.6 Uploading a package

5.6.1 Uploading to `ftp-master`

To upload a package, you need a personal account on `ftp-master.debian.org`, which you should have as an official maintainer. If you use `scp` or `rsync` to transfer the files, place them into `/org/ftp.debian.org/incoming/`; if you use anonymous FTP to upload, place them into `/pub/UploadQueue/`.

If you want to use the feature described in 'Delayed incoming' on page 25, you'll have to upload to `ftp-master`. It is the only upload point that supports delayed incoming.

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. If you don't want to bother with transferring the changes file last, you can simply copy your files to a temporary directory on `ftp-master` and then move them to `/org/ftp.debian.org/incoming/`.

Note: Do not upload to `ftp-master` cryptographic packages which belong to *contrib* or *non-free*. Uploads of such software should go to `non-us` (see 'Uploading to non-US' on this page). Furthermore packages containing code that is patent-restricted by the United States government can not be uploaded to `ftp-master`; depending on the case they may still be uploaded to `non-US/non-free` (it's in `non-free` because of distribution issues and not because of the license of the software). If you can't upload it to `ftp-master`, then neither can you upload it to the overseas upload queues on `chiark` or `erlangen`. If you are not sure whether U.S. patent controls or cryptographic controls apply to your package, post a message to `<debian-devel@lists.debian.org>` and ask.

You may also find the Debian packages 'dupload' on page 86 or 'dput' on page 86 useful when uploading packages. These handy programs help automate the process of uploading packages into Debian.

After uploading your package, you can check how the archive maintenance software will process it by running `dinstall` on your changes file:

```
dinstall -n foo.changes
```

Note that `dput` can do this for you automatically.

5.6.2 Uploading to non-US

As discussed above, export controlled software should not be uploaded to `ftp-master`. Instead, upload the package to `non-us.debian.org`, placing the files in `/org/non-us.debian.org/incoming/` (again, both 'dupload' on page 86 and 'dput' on page 86 can do this for you if invoked properly). By default, you can use the same account/password that works on `ftp-master`. If you use anonymous FTP to upload, place the files into `/pub/UploadQueue/`.

You can check your upload the same way it's done on `ftp-master`, with:

```
dinstall -n foo.changes
```

Note that U.S. residents or citizens are subject to restrictions on export of cryptographic software. As of this writing, U.S. citizens are allowed to export some cryptographic software, subject to notification rules by the U.S. Department of Commerce. However, this restriction has been waived for

software which is already available outside the U.S. Therefore, any cryptographic software which belongs in the *main* section of the Debian archive and does not depend on any package outside of *main* (e.g., does not depend on anything in *non-US/main*) can be uploaded to `ftp-master` or its queues, described above.

Debian policy does not prevent upload to non-US by U.S. residents or citizens, but care should be taken in doing so. It is recommended that developers take all necessary steps to ensure that they are not breaking current US law by doing an upload to non-US, *including consulting a lawyer*.

For packages in *non-US/main*, *non-US/contrib*, developers should at least follow the procedure outlined by the US Government (<http://www.bxa.doc.gov/Encryption/PubAvailEncSourceCodeNotify.html>). Maintainers of *non-US/non-free* packages should further consult the rules on notification of export (<http://www.bxa.doc.gov/Encryption/>) of non-free software.

This section is for information only and does not constitute legal advice. Again, it is strongly recommended that U.S. citizens and residents consult a lawyer before doing uploads to non-US.

5.6.3 Uploads via `chiark`

If you have a slow network connection to `ftp-master`, there are alternatives. One is to upload files to Incoming via a upload queue in Europe on `chiark`. For details connect to <ftp://ftp.chiark.greenend.org.uk/pub/debian/private/project/README.how-to-upload>.

Note: Do not upload packages containing software that is export-controlled by the United States government to the queue on `chiark`. Since this upload queue goes to `ftp-master`, the prescription found in ‘Uploading to `ftp-master`’ on page 37 applies here as well.

The program `dupload` comes with support for uploading to `chiark`; please refer to the documentation that comes with the program for details.

5.6.4 Uploads via `erlangen`

Another upload queue is available in Germany: just upload the files via anonymous FTP to <ftp://ftp.uni-erlangen.de/pub/Linux/debian/UploadQueue/>.

The upload must be a complete Debian upload, as you would put it into `ftp-master`’s Incoming, i.e., a `.changes` files along with the other files mentioned in the `.changes`. The queue daemon also checks that the `.changes` is correctly signed with GnuPG or OpenPGP by a Debian developer, so that no bogus files can find their way to `ftp-master` via this queue. Please also make sure that the Maintainer field in the `.changes` contains *your* e-mail address. The address found there is used for all replies, just as on `ftp-master`.

There’s no need to move your files into a second directory after the upload, as on `chiark`. And, in any case, you should get a mail reply from the queue daemon explaining what happened to

your upload. Hopefully it should have been moved to `ftp-master`, but in case of errors you're notified, too.

Note: Do not upload packages containing software that is export-controlled by the United States government to the queue on `erlangen`. Since this upload queue goes to `ftp-master`, the prescription found in 'Uploading to `ftp-master`' on page 37 applies here as well.

The program `dupload` comes with support for uploading to `erlangen`; please refer to the documentation that comes with the program for details.

5.6.5 Other upload queues

Another upload queue is available which is based in the US, and is a good backup when there are problems reaching `ftp-master`. You can upload files, just as in `erlangen`, to <ftp://samosa.debian.org/pub/UploadQueue/>.

An upload queue is available in Japan: just upload the files via anonymous FTP to <ftp://master.debian.or.jp/pub/Incoming/upload/>.

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

5.7 Determining section and priority of a package

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` 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(8)` and <http://www.debian.org/Bugs/Developer#maintincorrect>.

Note also that the term “section” is used for the separation of packages according to their licensing, e.g. *main*, *contrib* and *non-free*. This is described in another section, ‘The Debian archive’ on page 17.

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 75), how to update them and reorder them, and how to process and close them.

The bug tracking system's features interesting to developers 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 in 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", that 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 44.

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 interesting to developers 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 75 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 to 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 may either ask for help on `<debian-devel@lists.debian.org>` or reassign it to `debian-policy` to let them decide which package is at fault.

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 The bug submitter may have forgotten to provide some information, in which case you have to ask them 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.
- 4 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 that you send at the same time to the author. Make sure to send the patch in the BTS and to tag the bug as `patch`.

- 5 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.
- 6 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 your packages, it is your responsibility as the package maintainer to close the bug. However, you should not close the 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.

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+)*\s*/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`.

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 an `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 your bug number.

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 68.

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 fix 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. Useful information includes, for example:

- What 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 page 47 first)
- Any assistance you can provide to help with testing (exploits, regression testing, etc.)
- Any information needed for the advisory (see 'Security Advisories' on the following 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 a few ways a developer can learn of a security problem:

- he notices 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 contacts 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 83).

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*!
- 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 85 and ‘`debootstrap`’ on page 85).

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.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 the proper *distribution*-proposed-updates on `ftp-master` or in the non-US archive.

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 in 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). 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.

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 ‘Determining section and priority of a package’ on page 40.

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

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

that the package be removed. 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.

You also have to detail the reasons justifying that 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.

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. 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 more 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 the others about that, 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 `0: package -- short description` indicating that the package is now orphaned. The severity of the bug should be set to *normal*. 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 the package is especially crucial to Debian, you should instead submit a bug against `wnpp` and title it `RFA: package -- short description` and set its severity to *important*. RFA stands for *Request For Adoption*. Definitely copy the message to `debian-devel` in this case, as described above.

Read instructions on the WNPP web pages (<http://www.debian.org/devel/wnpp/>) for more information.

5.9.5 Adopting a package

A list of packages in need of a new maintainer is available at 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 78.

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 27 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, doing 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 85). 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 85).

If you can’t set up a proper chroot, `dpkg-depcheck` may be of assistance (see ‘`dpkg-depcheck`’ on page 87).

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 already being installed (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 source NMU’ on page 58 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 -mporter-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`.

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). 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 the third-level number on the Debian part of the version. For instance, if the latest version you are recompiling against was version "2.9-3", your NMU should carry a version of "2.9-3.0.1". If the latest version was "3.4-2.1", your NMU should have a version number of "3.4-2.1.1".

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 56, 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.

However, 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.)

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 compilers 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 the 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.

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 88.

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 85. 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.

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.

Debian porters, who compile packages for different architectures, occasionally do binary-only NMUs as part of their porting activity (see ‘Porting and being ported’ on page 52). Another reason why NMUs are done is when a Debian developer needs to fix another developer’s packages in order to address serious security problems or crippling bugs, especially during the freeze, or when the package maintainer is unable to release a fix in a timely fashion.

This chapter contains information providing guidelines for when and how NMUs should be done. A fundamental distinction is made between source and binary-only NMUs, which is explained in the next section.

5.11.1 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 by 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. In this chapter, if we use the unqualified term "NMU", we refer to any type of non-maintainer upload NMUs, whether source and binary, or binary-only.

5.11.2 Who can do an NMU

Only official, registered Debian maintainers can do binary or source NMUs. An official maintainer 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.

5.11.3 When to do a source NMU

Guidelines for when to do a source NMU depend on the target distribution, i.e., stable, unstable, or experimental. Porters have slightly different rules than non-porters, due to their unique circumstances (see 'When to do a source NMU if you are a porter' on page 54).

When a security bug is detected, the security team may do an NMU. Please refer to 'Handling security-related bugs' on page 45 for more information.

During the release cycle (see 'Stable, testing, and unstable' on page 21), NMUs which fix serious or higher severity bugs are encouraged and accepted. Even during this window, however, you should endeavor to reach the current maintainer of the package; they might be just about to upload a fix for the problem. As with any source NMU, the guidelines found in 'How to do a source NMU' on the next page need to be followed. Special exceptions are made for 'Bug squashing parties' on page 77.

Uploading bug fixes to unstable by non-maintainers should only be done by following this protocol:

- 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 'How to do a source NMU' on the current page, test it carefully on your machine (cf. 'Testing the package' on page 34). 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 incoming' on page 25), 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 27 (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.

5.11.4 How to do a source NMU

The following applies to porters insofar as they are playing the dual role of being both package bug-fixers and package porters. If a porter has to change the Debian source archive, their upload is automatically a source NMU and is subject to its rules. If a porter is simply uploading a recompiled binary package, the rules are different; see 'Guidelines for porter uploads' on page 53.

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.

Source 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'. 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'.

Source NMUs must have a new changelog entry

A non-maintainer 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 non-maintainer's email address in the log entry and the NMU version number in it.

By convention, source NMU changelog entries start with the line

```
* Non-maintainer upload
```

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 54 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.

If the source NMU (non-maintainer upload) fixes some existing bugs, these bugs should be tagged *fixed* in the Bug Tracking System rather than closed. By convention, only the official package maintainer or the original bug submitter are allowed to close bugs. Fortunately, Debian's archive

system recognizes NMUs and thus marks the bugs fixed in the NMu appropriately if the person doing the NMu has listed all bugs in the changelog with the `Closes: bug#nnnnn` syntax (see ‘When bugs are closed by new uploads’ on page 44 for more information describing how to close bugs via the changelog). Tagging the bugs *fixed* ensures that everyone knows that the bug was fixed in an NMu; however the bug is left open until the changes in the NMu are incorporated officially into the package by the official package maintainer.

Also, after doing an NMu, you have to open a new bug and include a patch showing all the changes you have made. Alternatively you can send that information to the existing bugs that are fixed by your NMu. 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.

Building source NMUs

Source NMu packages are built normally. Pick a distribution using the same rules as found in ‘Picking a distribution’ on page 36, follow the other prescriptions in ‘Uploading a package’ on page 37.

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.

5.11.5 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. You can either 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 and 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 facing page).

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 time. It is strongly recommended that packages in which 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`.
- 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 27), the co-maintainers should subscribe themselves to the appropriate source package.

Collaborative maintenance can often be further eased with the use of tools on Alioth (see ‘Debian *Forge: Alioth’ on page 31).

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 lets 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`, 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 81 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 hand-crafted 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 84), 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://people.debian.org/~srivasta/rules/>.

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 bug directly in the source, if 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.

`db`s 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 `db`s for more information and `hello-db`s 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., if the user can install only the subset she needs, 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 67 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 to the foo server”)?

Be careful to avoid spelling and grammar mistakes. Ensure that you spell-check it. `ispell` has a special `-g` option for `debian/control` files:

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

6.2.4 Upstream home page

We recommend that you add the URL for the package’s home page to the package description in `debian/control`. This information should be added at the end of description, using the following format:

```
.  
Homepage: http://some-project.some-place.org/
```

Note the spaces prepending the line, which serves to break the lines correctly. To see an example of how this displays, see <http://packages.debian.org/unstable/text/docbook-dsssl.html>.

If there is no home page for the software, this should naturally be left out.

Note that we expect this field will eventually be replaced by a proper `debian/control` field understood by `dpkg` and `packages.debian.org`. If you don’t want to bother migrating the home page from the description to this field, you should probably wait until that is available.

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 44 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 44.

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 42 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, for instance, in a changelog entry like this:

```
* Maintainer upload, closes: #42345, #44484, #42444.
```

This will close the NMU bugs tagged “fixed” when the package makes it into the archive. The bug for the fact that an NMU was done can be closed the same way. Of course, it’s also perfectly acceptable to close NMU-fixed bugs by other means; see ‘Responding to bugs’ on page 42.

6.3.3 Common errors in changelog entries

The following examples demonstrate some common errors or example 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.

6.4 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 sh-bang` 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 the 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`.

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, it's 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=HEAD&content-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. A 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 quality output 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 `Package` 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 82) when run over the entire Debian archive.

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 that 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

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. 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.

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 82) as possible. If you do not find that possible, then you should consider orphaning some of your packages (see ‘Orphaning a package’ on page 51). Alternatively, you may ask the help of other people in order to catch up 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 61).

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 announce explains what area will be focused on during the party: usually they focus on release critical bugs but it may happen that they decide to help finish a major upgrade going on (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 announce of the party is considered like a 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 applies as usually, they should send the patch of the NMU in the BTS (in one of the open bugs fixed by the NMU or in a new bug tagged fixed). They should also respect the maintainer's wishes if he expressed some.

If someone doesn't feel confident with an NMU, he should 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 27. You can do so by using the <package-name>@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.

The first step is to politely contact the maintainer, and wait for a response, for a reasonable time. 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 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's the last time a developer has 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.

One big 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 should 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.

Once you have gathered all of this, you can contact `<debian-qa@lists.debian.org>`. People on this alias will use the information you provided in order to decide how to proceed. For example, they might orphan one or all of the packages of the maintainer. If a packages 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.

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.

If you wish to volunteer as a sponsor, you can sign up at <http://www.internatif.org/bortzmeyer/debian/sponsor/>.

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 can not 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.

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 27.

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.

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 help convenience 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 with 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 manipulated packages; as such, they are required 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 modularly.

You can find documentation for this package in the `debconf-doc` package.

Many feel that this system should be used for all packages requiring interactive configuration; see ‘Configuration management with `debconf`’ on page 71. `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 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 with their packages.

A.2.1 `lintian`

`lintian` dissects Debian packages and emits information on 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 is its basis in Policy, and commonly how you can fix the problem.

Refer to ‘Testing the package’ on page 34 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/>. Those reports contain the latest `lintian` output on 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 86) 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’s 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 63 for more information on why these might or might not be desired.

A.3.1 `debhelper`

`debhelper` is a collection of programs that can be used in `debian/rules` to automate common tasks related to building binary Debian packages. Programs are included to install various files into your package, compress files, fix file permissions, integrate your package with the Debian menu system.

Unlike some approaches, `debhelper` is broken into several small, granular commands which act in a consistent manner. As such, it allows a greater granularity of 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 pre-cursor to debhelper, is a less granular `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. However, it's not a bug to use debmake.

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.

Note that yada is called "essentially unmaintained" by its own maintainer, Charles Briscoe-Smith. As such, it can be considered deprecated.

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 file-system. 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 depends. 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 build doing the build within User-mode-linux.

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 55 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.6 Maintenance automation

The following tools help automate different maintenance tasks, from adding changelog entries or signature lines, looking up bugs in Emacs, to making use of the newest and official use of `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. The `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`

Contains best practices for people maintaining packages that 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 that are installed on your system but no longer available elsewhere, or to store 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, when editing `debian/changelog`, there are handy functions for finalizing a version and listing the package's current bugs.

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.