Debian Developer's Reference

Release 14.14

Developer's Reference Team

2026-06-25

CONTENTS

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

This is Debian Developer's Reference version `14.14`, released on `2026-06-25`.

If you want to print this reference, you should use the pdf version. This manual is also available in some other languages.

CHAPTER

ONE

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 and maintainers.

The procedures discussed within include how to become a member (*Applying to Become a Member*); how to create new packages (*New packages*) and how to upload packages (*Uploading a package*); how to handle bug reports (*Handling bugs*); how to move, remove, or orphan packages (*Moving, removing, renaming, orphaning, adopting, and reintroducing packages*); how to port packages (*Porting and being ported*); and how and when to do interim releases of other maintainers' packages (*Non-Maintainer Uploads (NMUs)*).

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

It should be clear that this reference does not discuss the technical details of Debian packages nor how to generate them. Nor does this reference detail the standards to which Debian software must comply. All of such information can be found in the Debian Policy Manual.

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

CHAPTER

TWO

APPLYING TO BECOME A MEMBER

2.1 Getting started

So, you've read all the documentation, you've gone through the Debian New Maintainers' Guide (or its successor, Guide for Debian Maintainers), 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*. `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* for details. The IRC channel `#debian` can also be helpful; see *IRC channels*.

When you know how you want to contribute to Debian, 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* and https://wiki.debian.org/DebianMentorsFaq for more information on sponsoring). On the other hand, if you are interested in porting Debian to alternative architectures or kernels you can subscribe to port specific mailing lists and ask there how to get started. Finally, if you are interested in documentation or Quality Assurance (QA) work you can join maintainers already working on these tasks and submit patches and improvements.

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

2.2 Debian mentors and sponsors

The mailing list `debian-mentors@lists.debian.org` has been set up for novice maintainers who seek help with initial packaging and other developer-related issues. Every new developer is invited to subscribe to that list (see *Mailing lists* 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 member application to go through, you might be able find a sponsor to upload your package for you. Sponsors are people who are official Debian Developers, and who are willing to criticize and upload your packages for you. Please read the debian-mentors FAQ at https://wiki.debian.org/DebianMentorsFaqfirst.

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

2.3 Registering as a Debian member

Before you decide to register with Debian, you will need to read all the information available at the New Members Corner. It describes in detail the preparations you have to do before you can register to become a Debian member. For example, before you apply, you have to read the Debian Social Contract. Registering as a member means that you agree with and pledge to uphold the Debian Social Contract; it is very important that member are in accord with the essential ideas behind Debian. Reading the GNU Manifesto would also be a good idea.

The process of registering as a member is a process of verifying your identity and intentions, and checking your technical skills. As the number of people working on Debian has grown to over 1000 and our systems are used in several very important places, we have to be careful about being compromised. Therefore, we need to verify new members before we can give them accounts on our servers and let them upload packages.

Before you actually register you should have shown that you can do competent work and will be a good contributor. You show this by submitting patches through the Bug Tracking System and having a package sponsored by an existing Debian Developer for a while. Also, we expect that contributors are interested in the whole project and not just in maintaining their own packages. If you can help other maintainers by providing further information on a bug or even a patch, then do so!

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

If you do not have an OpenPGP key yet, generate one. Every developer needs an OpenPGP key in order to sign and verify package uploads. You should read the manual for the software you are using, since it has much important information that 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* for more information on maintaining your public key.

Debian uses the `GNU Privacy Guard` (package `gnupg` version 2 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 9580.

Your key length must be greater than 2048 bits (4096 bits is preferred); there is no reason to use a smaller key, and doing so would be much less secure.

If your public key isn't on a public key server such as `subkeys.pgp.net`, please read the documentation available at NM Step 2: Identification. That document contains instructions on how to put your key on the public key servers. The New Maintainer Group will put your public key on the servers if it isn't already there.

Some countries restrict the use of cryptographic software by their citizens. This need not impede one's activities as a Debian package maintainer however, as it may be perfectly legal to use cryptographic products for authentication, rather than encryption purposes. If you live in 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 member, you need an existing Debian Developer to support your application (an `advocate`). After you have contributed to Debian for a while, and you want to apply to become a registered developer, an existing developer with whom you have worked over the past months has to express their belief that you can contribute to Debian successfully.

When you have found an advocate, have your OpenPGP key signed and have already contributed to Debian for a while, you're ready to apply. You can simply register on our application page. 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 Member process. You can always check your status on the applications status board.

For more details, please consult New Members Corner at the Debian web site. Make sure that you are familiar with the necessary steps of the New Member process before actually applying. If you are well prepared, you can save a

lot of time later on.

CHAPTER

THREE

DEBIAN DEVELOPER'S DUTIES

3.1 Package Maintainer's Duties

As a package maintainer, you're supposed to provide high-quality packages that are well integrated into the system and that adhere to the Debian Policy.

3.1.1 Work towards the next `stable` release

Providing high-quality packages in `unstable` is not enough; most users will only benefit from your packages when they are released as part of the next `stable` release. You are thus expected to collaborate with the release team to ensure your packages get included.

More concretely, you should monitor whether your packages are migrating to `testing` (see *The testing distribution*). When the migration doesn't happen after the test period, you should analyze why and work towards fixing this. It might mean fixing your package (in the case of release-critical bugs or failures to build on some architecture) but it can also mean updating (or fixing, or removing from `testing`) other packages to help complete a transition in which your package is entangled due to its dependencies. The release team might provide you some input on the current blockers of a given transition if you are not able to identify them.

3.1.2 Maintain packages in `stable`

Most of the package maintainer's work goes into providing updated versions of packages in `unstable`, but their job also entails taking care of the packages in the current `stable` release.

While changes in `stable` are discouraged, they are possible. Whenever a security problem is reported, you should collaborate with the security team to provide a fixed version (see *Handling security-related bugs*). When bugs of severity important (or more) are reported against the `stable` version of your packages, you should consider providing a targeted fix. You can ask the `stable` release team whether they would accept such an update and then prepare a `stable` upload (see *Special case: uploads to the stable and oldstable distributions*).

3.1.3 Manage release-critical bugs

Generally you should deal with bug reports on your packages as described in *Handling bugs*. 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` make the package unsuitable for inclusion in the next `stable` release. They can thus delay the Debian release (when they affect a package in `testing`) or block migrations to `testing` (when they only affect the package in `unstable`). In the worst scenario, they will lead to the package's removal. That's why these bugs need to be corrected as quickly as possible.

If, for any reason, you aren't able fix an RC bug in a package of yours within 2 weeks (for example due to time constraints, or because it's difficult to fix), you should mention it clearly in the bug report and you should tag the bug `help` to invite other volunteers to chime in. Be aware that RC bugs are frequently the targets of Non-Maintainer Uploads (see *Non-Maintainer Uploads (NMUs)*) because they can block the `testing` migration of many packages.

Lack of attention to RC bugs is often interpreted by the QA team as a sign that the maintainer has disappeared without properly orphaning their package. The MIA team might also get involved, which could result in your packages being orphaned (see *Dealing with inactive and/or unreachable maintainers*).

3.1.4 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. These bug reports should be forwarded to the upstream developers so that they can be fixed in a future upstream release. Usually it is best if you can do this, but alternatively, you may ask the bug submitter to do it. Ideally, you also get upstream to *subscribe* for Debian Package Tracker notifications for their software in Debian.

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.

In cases where a bug report is forwarded upstream, it may be helpful to remember that the bts-link service can help with synchronizing states between the upstream bug tracker and the Debian one.

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.

As most upstreams nowadays use git for version control, in most cases git-buildpackage offers the most convenient way to create and maintain patches in Debian that so they are submit upstream. For details, see git-buildpackage man pages about using `pq` to write and test `debian/patches` as git commits, and having git remote `upstreamvcs` to easily cherry-pick patches to and from upstream git branches.

If you find that the upstream developers are or become hostile towards Debian or the free software community, you may want to re-consider the need to include the software in Debian. Sometimes the social cost to the Debian community is not worth the benefits the software may bring.

3.2 Administrative Duties

A project of the size of Debian relies on some administrative infrastructure to keep track of everything. As a project member, you have some duties to ensure everything keeps running smoothly.

3.2.1 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*.

3.2.2 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*). Back your keys up; keep a copy offline. Read the documentation that comes with your software; read the PGP FAQ and OpenPGP Best Practices.

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

If you add signatures to your public key, or add user identities, you can update the Debian key ring by sending your key to the key server at `keyring.debian.org`. Updates are processed at least once a month by the `debian-keyring` package maintainers.

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

The same key extraction routines discussed in *Registering as a Debian member* apply.

You can find a more in-depth discussion of Debian key maintenance in the documentation of the `debian-keyring` package and the https://keyring.debian.org/ site.

3.2.3 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.

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 OpenPGP-signed/encrypted email messages.

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

3.2.4 Decision making

Besides the issues covered in *Voting*, Debian is closer to a do-ocracy, than a democracy. Generally speaking, this means that decisions are taken by the people who do the work, which in most cases is package maintainers and DPL delegates. Public discussion:

- while encouraged and may be appreciated as input to take a decision,
- is not required to take a decision,
- nor can overrule a decision (the latter is possible via a General Resolution or by escalating to the Debian Technical Committee).

In the spirit of do-ocracy, members of the Debian community are expected to contribute in concrete and constructive ways, rather than demand that others do any kind of work for them.

For more on the social aspects of contributing to Debian, you are strongly encouraged to read the Debian Community Guidelines at http://people.debian.org/~enrico/dcg/

3.2.5 Going on vacation gracefully

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

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

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

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

Ideally, you should sign up at the OpenPGP coordination pages when booking a holiday and check if anyone there is looking for signing. This is especially important when people go to exotic places where we don't have any developers yet but where there are people who are interested in applying.

3.2.6 Retiring

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

- Orphan all your packages, as described in *Orphaning a package*.

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

- Remove yourself from uploaders for co- or team-maintained packages.
- If you received mails via a @debian.org e-mail alias (e.g. press@debian.org) and would like to get removed, open a RT ticket for the Debian System Administrators. Just send an e-mail to `admin@rt.debian.org` with "Debian RT" somewhere in the subject stating from which aliases you'd like to get removed.
- Please remember to also retire from teams, e.g. remove yourself from team wiki pages or salsa groups.
- Use the link https://nm.debian.org/process/emeritus to log in to nm.debian.org, request emeritus status and write a goodbye message that will be automatically posted on debian-private.

  Salsa is used as authentification provider to the NM site, in case you run into problems opening the retirement process yourself, contact NM front desk using `nm@debian.org`

It is important that the above process is followed, because finding inactive developers and orphaning their packages takes significant time and effort.

3.2.7 Returning after retirement

A retired developer's account is marked as "emeritus" when the process in *Retiring* is followed, and "removed" otherwise. Retired developers with an "emeritus" account can get their account re-activated as follows:

- Log on onto nm.debian.org using your salsa credentials.
- Follow the Return from Emeritus Wizard.
- Go through a shortened NM process (to ensure that the returning developer still knows important parts of P&P and T&S).

In case your run into problems contact `nm@debian.org` for further instructions.

Retired developers with a "removed" account need to go through full NM again.

CHAPTER

FOUR

RESOURCES FOR DEBIAN MEMBERS

In this chapter you will find a very brief roadmap of the Debian mailing lists, the Debian machines which may be available to you as a member, and all the other resources that are available to help you in your 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`. 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 https://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 for more information. The Debian Community Guidelines are also worth reading.

4.1.2 Core development mailing lists

The core Debian mailing lists that developers should use are:

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

There are other mailing lists available for a variety of special topics; see https://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 `master.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) 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.

4.2 IRC channels

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

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

The main channel for Debian development is `#debian-devel`. It is a very active channel; it will typically have a minimum of 150 people at any time of day. It's a channel for people who work on Debian, it's not a support channel (there's `#debian` for that). It is however open to anyone who wants to lurk (and learn). Its topic is commonly full of interesting information for developers.

Since `#debian-devel` is an open channel, you should not speak there of issues that are discussed in `debian-private@lists.debian.org`. There's another channel for this purpose, it's called `#debian-private` and it's protected by a key. This key is available at `master.debian.org:~debian/misc/irc-password`.

There are other additional channels dedicated to specific subjects. `#debian-bugs` is used for coordinating bug squashing parties. `#debian-boot` is used to coordinate the work on the debian-installer. `#debian-doc` is occasionally used to talk about documentation, like the document you are reading. Other channels are dedicated to an architecture or a set of packages: `#debian-kde`, `#debian-dpkg`, `#debian-perl`, `#debian-python`...

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.

4.3 Documentation

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

4.4 Debian machines

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

Some of the machines are available for individual developers to use, as long as the developers follow the rules set forth in the Debian Machine Usage Policies.

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 https://db.debian.org/machines.cgi. That web page contains machine names, contact information, information about who can log in, SSH keys etc.

If you have a problem with the operation of a Debian server, and you think that the system operators need to be notified of this problem, you can check the list of open issues in the DSA (Debian System Administration) Team's queue of our request tracker at https://rt.debian.org/ (you can login with user "debian", its password is available at `master.debian.org:~debian/misc/rt-password`). To report a new problem in the request tracker, simply send a mail to `admin@rt.debian.org` and make sure to put the string "Debian RT" somewhere in the subject. To contact the DSA team by email, use `dsa@debian.org` for anything that contains private or privileged information and should not be made public, and `debian-admin@lists.debian.org` otherwise. The DSA team is also present on the `#debian-admin` IRC channel on OFTC.

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* for information on how to submit bugs.

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

4.4.1 The bugs server

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

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

4.4.2 The ftp-master server

The `ftp-master.debian.org` server holds the canonical copy of the Debian archive. Generally, packages uploaded to ftp.upload.debian.org end up on this server; see *Uploading a package*.

It is restricted; a mirror is available on `mirror.ftp-master.debian.org`.

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, orphaning, adopting, and reintroducing packages*.

4.4.3 The www-master server

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

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

4.4.4 The people web server

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

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

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

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

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

4.4.5 salsa.debian.org: Git repositories and collaborative development platform

If you want to use a git repository for any of your Debian work, you can use Debian's GitLab instance called Salsa for that purpose. Gitlab provides also the possibility to have merge requests, wiki pages, bug trackers among many other services as well as a fine-grained tuning of access permission, to help working on projects collaboratively.

For more information, please see the documentation at https://wiki.debian.org/Salsa/Doc.

Any Debian package hosted on Salsa has also access to the Salsa CI . The Salsa CI pipeline mimics the tests that are run after each upload to Debian, but instead of having to wait for results or risk the health of the Debian repositories, Salsa CI provides you with instant feedback about any problems the changes you made may have created or solved.

4.4.6 GitHub.com: Submitting pull requests to upstream repositories

If some upstream repository is hosted on GitHub.com, you can use the Debian organization to create repository forks and submit changed branches with pull requests to upstream maintainers.

The organization is open to all Debian Members. To request membership, open an issue in the Debian/.github meta repository.

4.4.7 chroots to different distributions

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

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

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

4.5 The Developers Database

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

Developers can log into the database to change various information about themselves, such as:

- forwarding address for your debian.org email as well as spam handling. See https://db.debian.org/forward.html for a description of all the options.
- 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, 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 https://db.debian.org/doc-general.html.

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

4.6 The Debian archive

The Debian distribution consists of a lot of packages (currently around `40000` source packages) 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-amd64/
dists/stable/main/binary-armel/
dists/stable/main/binary-i386/
     ...
```

(continues on next page)

(continued from previous page)

```
dists/stable/main/source/
     ...
dists/stable/main/disks-amd64/
dists/stable/main/disks-armel/
dists/stable/main/disks-i386/
     ...

dists/stable/contrib/
dists/stable/contrib/binary-amd64/
dists/stable/contrib/binary-armel/
dists/stable/contrib/binary-i386/
     ...
dists/stable/contrib/source/

dists/stable/non-free/
dists/stable/non-free/binary-amd64/
dists/stable/non-free/binary-armel/
dists/stable/non-free/binary-i386/
     ...
dists/stable/non-free/source/

dists/stable/non-free-firmware/
dists/stable/non-free-firmware/binary-amd64/
dists/stable/non-free-firmware/binary-armel/
dists/stable/non-free-firmware/binary-i386/
     ...
dists/stable/non-free-firmware/source/

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

dists/unstable
dists/unstable/main/
     ...
dists/unstable/contrib/
     ...
dists/unstable/non-free/
     ...
dists/unstable/non-free-firmware/
     ...

pool/
pool/main/a/
pool/main/a/apt/
     ...
pool/main/b/
pool/main/b/bash/
     ...
pool/main/liba/
```

(continues on next page)

(continued from previous page)

```
pool/main/liba/libalias-perl/
     ...
pool/main/m/
pool/main/m/mailx/
     ...
pool/non-free/d/
pool/non-free/d/doc-rfc/
     ...
pool/non-free-firmware/f/
pool/non-free-firmware/f/firmware-nonfree/
     ...
```

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 four directories, namely `main`, `contrib`, `non-free` and `non-free-firmware`.

In each of the areas, there is a directory for the source packages (`source`) and a directory for each supported architecture (`binary-i386`, `binary-amd64`, 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-amd64`, etc.).

4.6.1 Sections

The `main` section of the Debian archive is what makes up the **official Debian 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.

Every package in the main section must fully comply with the Debian Free Software Guidelines (DFSG) and with all other policy requirements as described in the Debian Policy Manual. 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` or `non-free-firmware` sections. These packages are not considered as part of the Debian distribution, though we enable their use, and we provide infrastructure (such as our bug-tracking system and mailing lists) for these non-free software packages.

The Debian Policy Manual contains a more exact definition of the four sections. The above discussion is just an introduction.

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

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

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

4.6.2 Architectures

In the first days, the Linux kernel was only available for Intel i386 (or greater) platforms, and so was Debian. But as Linux became more and more popular, the kernel was ported to other architectures and Debian started to support them. And as if supporting so much hardware was not enough, Debian decided to build some ports based on other Unix kernels, like `hurd` and `kfreebsd`.

Debian GNU/Linux 1.3 was only available as `i386`. Debian 2.0 shipped for `i386` and `m68k` architectures. Debian 2.1 shipped for the `i386`, `m68k`, `alpha`, and `sparc` architectures. Since then Debian has grown hugely. Debian 9 supports a total of ten Linux architectures (`amd64`, `arm64`, `armel`, `armhf`, `i386`, `mips`, `mips64el`, `mipsel`, `ppc64el`, and `s390x`) and two kFreeBSD architectures (`kfreebsd-i386` and `kfreebsd-amd64`).

Information for developers and users about the specific ports are available at the Debian Ports web pages.

4.6.3 Packages

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

Depending on the format of the source package, it will consist of one or more files in addition to the mandatory `.dsc` file:

- with format "1.0", it has either a `.tar.gz` file or both an `.orig.tar.gz` and a `.diff.gz` file;
- with format "3.0 (quilt)", it has a mandatory `.orig.tar.{gz,bz2,xz}` upstream tarball, multiple optional `.orig-`*component*`.tar.{gz,bz2,xz}` additional upstream tarballs and a mandatory `debian.tar.{gz,bz2,xz}` debian tarball;
- with format "3.0 (native)", it has only a single `.tar.{gz,bz2,xz}` tarball.

If a package is developed specially for Debian and is not distributed outside of Debian, there is just one `.tar.{gz,bz2,xz}` file, which contains the sources of the program; it's called a "native" source package. If a package is distributed elsewhere too, the `.orig.tar.{gz,bz2,xz}` file stores the so-called `upstream source code`, that is the source code that's distributed by the `upstream maintainer` (often the author of the software). In this case, the `.diff.gz` or the `debian.tar.{gz,bz2,xz}` contains the changes made by the Debian maintainer.

The `.dsc` file lists all the files in the source package together with checksums (`md5sums`, `sha1sums`, `sha256sums`) 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 a mirror site. For instance, at the mirror site `ftp.us.debian.org` the Debian archive itself is contained in /debian, which is a common location (another is `/pub/debian`).

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

4.6.4.1 Stable, testing, and unstable

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

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

The *testing* distribution is generated automatically by taking packages from `unstable` if they satisfy certain criteria. Those criteria should ensure a good quality for packages within `testing`. The update to `testing` is launched twice each day, right after the new packages have been installed. See *The testing distribution*.

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

This development cycle is based on the assumption that the `unstable` distribution becomes `stable` after passing a period of being in `testing`. Even once a distribution is considered stable, a few bugs inevitably remain — that's why the stable distribution is updated every now and then. However, these updates are tested very carefully and have to be introduced into the archive individually to reduce the risk of introducing new bugs. You can find proposed additions to `stable` in the `proposed-updates` directory. Those packages in `proposed-updates` that pass muster are periodically moved as a batch into the stable distribution and the revision level of the stable distribution is incremented (e.g., '6.0' becomes '6.0.1', '5.0.7' becomes '5.0.8', and so forth). Please refer to *Special case: uploads to the stable and oldstable distributions* for details.

Note that development in `unstable` during the freeze should not be continued as usual, as packages are still build in `unstable`, before they migrate to `testing`, thus `unstable` should only contain packages meant for `testing`. Thus only upload to `unstable` during freezes, if you are planning to request an unblock (or if the package is not in `testing`).

If you want to develop new stuff for after the freeze, upload to `experimental` instead.

4.6.4.2 More information about the testing distribution

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

For more details, please see the *The testing distribution*.

4.6.4.3 Experimental

The `experimental` distribution is a special distribution. It is not a full distribution in the same sense as `stable`, `testing` 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://deb.debian.org/debian/ experimental main
deb-src http://deb.debian.org/debian/ experimental main
```

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

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

Some experimental software can still go into `unstable`, with a few warnings in the description, but that isn't recommended because packages from `unstable` are expected to propagate to `testing` and thus to `stable`. You should not be afraid to use `experimental` since it does not cause any pain to the ftpmasters, the experimental packages are periodically 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 11 is called `bullseye`; Debian 12, `bookworm`; Debian 13, `trixie`; the next release, Debian 14, will be called `forky` and Debian 15 will be called `duke`. 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. The codenames and versions for older releases are listed on the website.

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., `trixie`). 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 https://www.debian.org/mirror/. This useful page also includes information and tools which can be helpful if you are interested in setting up your own mirror, either for internal or public access.

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

4.8 The Incoming system

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

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

Once the package is accepted, the system sends a confirmation mail to the maintainer and closes all the bugs marked as fixed by the upload, and the auto-builders may start recompiling it. The package is now publicly accessible at

https://incoming.debian.org/ until it is really installed in the Debian archive. This happens four times a day (and is also called the `dinstall run` for historical reasons); the package is then removed from incoming and installed in the pool along with all the other packages. Once all the other updates (generating new `Packages` and `Sources` index files for example) have been made, a special script is called to ask all the primary mirrors to update themselves.

The archive maintenance software will also send the OpenPGP 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` or `debian-experimental-changes@lists.debian.org` instead.

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

4.9 Package information

4.9.1 On the web

Each package has several dedicated web pages. `https://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 `https://bugs.debian.org/`*package-name*.

4.9.2 The `dak ls` utility

`dak ls` is part of the dak suite of tools, listing available package versions for all known distributions and architectures. The `dak` tool is available on `ftp-master.debian.org`, and on the mirror on `mirror.ftp-master.debian.org`. It uses a single argument corresponding to a package name. An example will explain it better:

```
$ dak ls evince
evince     | 3.22.1-3+deb11u2 | oldstable           | source, amd64, arm64, armel,␣
↪armhf, i386, mips, mips64el, mipsel, ppc64el, s390x
evince     | 3.22.1-3+deb11u2 | oldstable-debug     | source
evince     | 3.30.2-3+deb12u1 | stable              | source, amd64, arm64, armel,␣
↪armhf, i386, mips, mips64el, mipsel, ppc64el, s390x
evince     | 3.30.2-3+deb12u1 | stable-debug        | source
evince     | 3.38.2-1         | testing             | source, amd64, arm64, armel,␣
↪armhf, i386, mips64el, mipsel, ppc64el, s390x
evince     | 3.38.2-1         | unstable            | source, amd64, arm64, armel,␣
↪armhf, i386, mips64el, mipsel, ppc64el, s390x
evince     | 3.38.2-1         | unstable-debug      | source
evince     | 40.4-1           | buildd-experimental | source, amd64, arm64, armel,␣
↪armhf, i386, mips64el, mipsel, ppc64el, s390x
evince     | 40.4-1           | experimental        | source, amd64, arm64, armel,␣
↪armhf, i386, mips64el, mipsel, ppc64el, s390x
evince     | 40.4-1           | experimental-debug  | source
```

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 all architectures. Each version of the package has been recompiled on all architectures.

4.10 The Debian Package Tracker

The Debian Package Tracker is an email and web-based tool to track the activity of a source package. You can get the same emails that the package maintainer gets, simply by *subscribing* to the package in the Debian Package Tracker.

The package tracker has a web interface at https://tracker.debian.org/ that puts together a lot of information about each source package. It features many useful links (BTS, QA stats, contact information, DDTP translation status, buildd logs) and gathers much more information from various places (30 latest changelog entries, testing status, etc.). It's a very useful tool if you want to know what's going on with a specific source package. Furthermore, once authenticated, you can subscribe and unsubscribe from any package with a single click.

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

For more in-depth information, you should have a look at its documentation. Among other things, it explains you how to interact with it by email, how to filter the mails that it forwards, how to configure your VCS commit notifications, how to leverage its features for maintainer teams, etc.

4.11 Developer's packages overview

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

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

4.12 Debian's FusionForge installation: Alioth

Until Alioth was deprecated and eventually turned off in June 2018, it was a Debian service based on a slightly modified version of the FusionForge software (which evolved from SourceForge and GForge). This software offered developers access to easy-to-use tools such as bug trackers, patch managers, project/task managers, file hosting services, mailing lists, VCS repositories, etc.

For many previously offered services replacements exist. This is important to know, as there are still many references to alioth which still need fixing. If you encounter such references please take the time to try fixing them, for example by filing bugs or when possible fixing the reference.

4.13 Goodies for Debian Members

Benefits available to Debian Members are documented on https://wiki.debian.org/MemberBenefits.

CHAPTER

FIVE

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) 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 for more information.

Assuming no one else is already working on your prospective package, you must then submit a bug report (*Bug reporting*) against the pseudo-package `wnpp` describing your plan to create a new package, including, but not limiting yourself to, the description of the package (so that others can review it), 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`. Please send a copy to `debian-devel@lists.debian.org` by using the X-Debbugs-CC header (don't use CC:, because that way the message's subject won't indicate the bug number). If you are packaging so many new packages (>10) that notifying the mailing list in separate messages is too disruptive, send a summary after filing the bugs to the debian-devel list instead. This will inform the other developers about upcoming packages and will allow a review of your description and package name.

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

If you think your package needs some explanations for the administrators of the NEW package queue, include them in your changelog, send to `ftpmaster@debian.org` a reply to the email you receive as a maintainer after your upload, or reply to the rejection email in case you are already re-uploading.

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

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

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

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

5.2 Recording changes in the package

Changes that you make to the package need to be recorded in the `debian/changelog` file, for human users to read and comprehend. 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 packaging 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*. 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*.

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

```
* New upstream release.
```

There are tools to help you create entries and finalize the `changelog` for release — see *devscripts* (command `dch`), *git-buildpackage* (command `gbp dch`) and *dpkg-dev-el*.

See also *Best practices for debian/changelog*.

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):

- 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*.
- Optionally run `debdiff` (see *debdiff*) to analyze changes from an older version, if one exists.
- Install the package and make sure the software works in an up-to-date `unstable` system.
- Upgrade the package from an older version to your new version.
- Remove the package, then reinstall it.
- Installing, upgrading and removal of packages can either be tested manually or by using the `piuparts` tool.
- With the package installed, run `adequate <package name>` to test its installed state for policy violations. Ideally, you should automate this step using `autopkgtest` (see *adequate*).
- Copy the source package in a different directory and try unpacking it and rebuilding it. This tests if the package relies on existing files outside of it, or if it relies on permissions being preserved on the files shipped inside the `.diff.gz` file.

5.4 Layout of the source package

There are two types of Debian source packages:

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

For the native packages, the source package includes a Debian source control file (`.dsc`) and the source tarball (`.tar.{gz,bz2,xz}`). A source package of a non-native package includes a Debian source control file, the original source tarball (`.orig.tar.{gz,bz2,xz}`) and the Debian changes (`.diff.gz` for the source format "1.0" or `.debian.tar.{gz,bz2,xz}` for the source format "3.0 (quilt)").

With source format "1.0", whether a package is native or not was determined by `dpkg-source` at build time. Nowadays it is recommended to be explicit about the desired source format by putting either "3.0 (quilt)" or "3.0 (native)" in `debian/source/format`. The rest of this section relates only to non-native packages.

The first time a version is uploaded that corresponds to a particular upstream version, the original source tar file must 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 current changelog entry has a different upstream version from the preceding entry. This behavior may be modified by using `-sa` to always include it or `-sd` to always leave it out.

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

Please notice that, in non-native packages, permissions on files that are not present in the `*.orig.tar.{gz,bz2,xz}` will not be preserved, as diff does not store file permissions in the patch. However, when using source format "3.0 (quilt)", permissions of files inside the `debian` directory are preserved since they are stored in a tar 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.

Packages are normally uploaded into `unstable`. Uploads to `unstable` or `experimental` should use these suite names in the changelog entry; uploads for other supported suites should use the suite codenames, as they avoid any ambiguity.

Actually, there are other possible distributions: *codename*`-security`, but read *Handling security-related bugs* for more information on those.

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

5.5.1 Special case: uploads to the `stable` and `oldstable` distributions

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

Uploads to a supported `stable` release should target their suite name in the changelog, i.e. `trixie` or `bookworm`. You should normally use `reportbug` and the `release.debian.org` pseudo-package to send a *source* `debdiff`, rationale and associated bug numbers to the stable release managers, and await a request to upload or further information.

If you are confident that the upload will be accepted without changes, please feel free to upload at the same time as filing the `release.debian.org` bug. However if you are new to the process, we would recommend getting approval before uploading so you get a chance to see if your expectations align with ours.

Either way, there must be an accompanying bug for tracking, and your upload must comply with these acceptance criteria defined by the the stable release managers. These criteria are designed to help the process be as smooth and frustration-free as possible.

- The bug you want to fix in `stable` must be fixed in `unstable` already (and not waiting in NEW or the delayed queue).
- The bug should be of severity "important" or higher.
- Bug meta-data - particularly affected versions - must be up to date.

- Fixes must be minimal and relevant and include a sufficiently detailed changelog entry.
- A source debdiff of the proposed change must be included in your request (not just the raw patches or "a debdiff can be found at $URL").
- The proposed package must have a correct version number (e.g. `...+deb13u1/...~deb13u1` for `trixie` or `+deb12u1/~deb12u1` for `bookworm`) and you should be able to explain what testing it has had. See the Debian Policy for the version number: https://www.debian.org/doc/debian-policy/ch-controlfields.html#special-version-conventions
- The update must be built in an `stable` environment or chroot (or `oldstable` if you target that).
- Fixes for security issues should be co-ordinated with the security team, unless they have explicitly stated that they will not issue an `DSA` for the bug (e.g. via a "no-dsa" marker in the *Debian Security Tracker*).
- Do not close `release.debian.org` bugs in debian/changelog. They will be closed by the release team once the package has reached the respective point release.

It is recommended to use `reportbug` as it eases the creation of bugs with correct meta-data. The release team makes extensive use of usertags to sort and manage requests and incorrectly tagged reports may take longer to be noticed and processed.

Uploads to the `oldstable` distributions are possible as long as it hasn't been archived. The same rules as for `stable` apply.

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 (`DSA`) are automatically copied to the appropriate `proposed-updates` archive when the advisory is released. See *Handling security-related bugs* for detailed information on handling security problems. If the security team deems the problem to be too benign to be fixed through a `DSA`, the stable release managers are usually willing to include your fix nonetheless in a regular upload to `stable`.

5.5.2 Special case: the `stable-updates` suite

Sometimes the stable release managers will decide that an update to stable should be made available to users sooner than the next scheduled point release. In such cases, they can copy the update to the `stable-updates` suite, use of which is enabled by the installer by default.

Initially, the process described in *Special case: uploads to the stable and oldstable distributions*. should be followed as usual. If you think that the upload should be released via `stable-updates`, mention this in your request. Examples of circumstances in which the upload may qualify for such treatment are:

- The update is urgent and not of a security nature. Security updates will continue to be pushed through the security archive. Examples include packages broken by the flow of time (c.f. `spamassassin` and the year 2010 problem) and fixes for bugs introduced by point releases.
- The package in question is a data package and the data must be updated in a timely manner (e.g. `tzdata`).
- Fixes to leaf packages that were broken by external changes (e.g. video downloading tools and `tor`).
- Packages that need to be current to be useful (e.g. `clamav`).
- Uploads to `stable-updates` should target their suite name in the changelog as usual, e.g. `trixie`.

Once the upload has been accepted to `proposed-updates` and is ready for release, the stable release managers will then copy it to the `stable-updates` suite and issue a Stable Update Announcement (`SUA`) via the `debian-stable-announce` mailing list.

Any updates released via `stable-updates` will be included in `stable` with the next point release as usual.

5.5.3 Special case: uploads to `testing/testing-proposed-updates`

Please see the information in the *Direct updates to testing* for details.

5.6 Uploading a package

5.6.1 Source and binary uploads

Each upload to Debian consists of a signed `.changes` file describing the requested change to the archive, plus the source and binary package files that are referenced by the `.changes` file.

If possible, the version of a package that is uploaded should be a source-only changes file. These are typically named `*_source.changes`, and reference the source package, but no binary `.deb` or `.udeb` packages. All of the corresponding architecture-dependent and architecture-independent binary packages, for all architectures, will be built automatically by the build daemons in a controlled and predictable environment (see *wanna-build* for more details).

For many source-only uploads you can use tag2upload instead, which means that you only need to push a signed Git tag to Salsa, instead of generating and signing `.dsc` and `.changes` files yourself. See https://wiki.debian.org/tag2upload.

There are several situations where a source-only upload is not possible.

The first upload of a new source package (see *New packages*) must include binary packages, so that they can be reviewed by the archive administrators before they are added to Debian.

If new binary packages are added to an existing source package, then the first upload that lists the new binary packages in `debian/control` must include binary packages, again so that they can be reviewed by the archive administrators before they are added to Debian. It is preferred for these uploads to be done via the `experimental` suite.

Uploads that will be held for review in other queues, such as packages being added to the `*-backports` suites, might also require inclusion of binary packages.

The build daemons will automatically attempt to build any `main` or `contrib` package for which the build-dependencies are available. Packages in `non-free` and `non-free-firmware` will not be built by the build daemons unless the package has been marked as suitable for auto-building (see *Marking non-free packages as auto-buildable*).

The build daemons only install build-dependencies from the `main` archive area. This means that if a source package has build-dependencies that are in the `contrib`, `non-free` or `non-free-firmware` archive areas, then uploads of that package need to include prebuilt binary packages for every architecture that will be supported. By definition this can only be the case for source packages that are themselves in the `contrib`, `non-free` or `non-free-firmware` archive areas.

Bootstrapping a new architecture, or a new version of a package with circular dependencies (such as a self-hosting compiler), will sometimes also require an upload that includes binary packages.

5.6.2 Uploading to `ftp-master`

To upload a package, you should upload the files (including the signed changes and dsc file) with anonymous ftp to `ftp.upload.debian.org` in the directory /pub/UploadQueue/. To get the files processed there, they need to be signed with a key in the Debian Developers keyring or the Debian Maintainers keyring (see https://wiki.debian.org/DebianMaintainer).

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

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

For removing packages or cancelling an upload, please see ftp://ftp.upload.debian.org/pub/UploadQueue/README and the Debian package *dcut*.

Finally, you should think about the status of your package with relation to `testing` before uploading to `unstable`. If you have a version in `unstable` waiting to migrate then it is generally a good idea to let it migrate before uploading another new version. You should also check the *The Debian Package Tracker* for transition warnings to avoid making uploads that disrupt ongoing transitions.

5.6.3 Delayed uploads

It is sometimes useful to upload a package immediately, but to want this package to arrive in the archive only a few days later. For example, when preparing a *Non-Maintainer Uploads (NMUs)*, you might want to give the maintainer a few days to react, while at the same time having the certainty that your upload will eventually take effect if the maintainer does not. A one-shot approach of (i) file bug with a patch (or Merge Request) and (ii) at once upload to DELAYED, is much more efficient than having to wait for days between the two steps.

An upload to the delayed directory keeps the package in the deferred uploads queue. When the specified waiting time is over, the package is moved into the regular incoming directory for processing. This is done through automatic uploading to `ftp.upload.debian.org` in upload-directory `DELAYED/`*X*`-day` (*X* between 0 and 15). 0-day is uploaded multiple times per day to `ftp.upload.debian.org`.

With dput, you can use the `--delayed` *DELAY* parameter to put the package into one of the queues.

Sometimes you'll need to remove a previously uploaded package from the DELAYED queue (e.g. because the package maintainer has asked you so). dcut(1) has examples of how to do so.

5.6.4 Security uploads

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

5.6.5 Other upload queues

There is an alternative upload queue in Europe at ftp://ftp.eu.upload.debian.org/pub/UploadQueue/. It operates in the same way as `ftp.upload.debian.org`, but should be faster for European developers.

Packages can also be uploaded via ssh to `ssh.upload.debian.org`; files should be put `/srv/upload.debian.org/UploadQueue`. This queue does not support *Delayed uploads*.

5.6.6 Notifications

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, `dak process-upload`. 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 some time to occur. Please be patient.

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

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

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

Also note that new uploads are announced on the *IRC channels* channel `#debian-devel-changes`. If your upload fails silently, it could be that your package is improperly signed, in which case you can find more explanations on `ssh.upload.debian.org:/srv/upload.debian.org/queued/run/log`.

5.7 Specifying the package section, subsection and priority

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

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

correct your `debian/control` file for your next upload, or else you may wish to make a change in the `override file`.

To alter the actual section that a package is put in, you need to first make sure that the `debian/control` file in your package is accurate. Next, 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. Use a Subject like `override: PACKAGE1:section/priority, [...], PACKAGEX:section/priority`, and include the justification for the change in the body of the bug report.

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

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

5.8 New upstream versions

To update a package for a new upstream release:

1. Read the upstream changelog, NEWS, and whatever other documentation they may have released with the new version.
2. If possible, inspect the full diff between the old and new upstream sources, potentially filtering out uninteresting parts using filterdiff (e.g. `filterdiff -x '*.po'`). If it's too big to review thoroughly, you can use `diffstat` to get a feel for the scope and nature of the changes (and thus where new bugs may appear), and to keep an eye for anything suspicious (e.g. unexpected use of network or the apperance of dubious binary blobs).
3. Port the old Debian packaging to the new version. This basically involves incrementing the `debian/changelog` and merging `debian/patches` from the old package to the new one.
4. Check to see if any bugs have been fixed that are currently open in the BTS. If they have been, close them in the `debian/changelog`.
5. If the patch/merge did not apply cleanly, figure out why. A patch may fail to apply if it's already been applied in the new upstream release, or if the upstream files the patch applies to have been substantially modified (or deleted).
6. If any changes were made to the build system (you'd know from steps 1 and 2), update the `debian/rules` and `debian/control` build dependencies if necessary.
7. Build the new package in an isolated environment, e.g. using `sbuild` or `pbuilder`. This ensures that all required build dependencies are listed in `debian/control` and eliminates the possibility of interference of any third party packages in your system.
8. Verify that the new package builds correctly and if so carry out the checks in _sanitycheck.

5.9 Handling bugs

Every developer has to be able to work with the Debian bug tracking system. This includes knowing how to file bug reports properly (see *Bug reporting*), how to update them and reorder them, and how to process and close them.

The bug tracking system's features are described in the BTS documentation for developers. This includes closing bugs, sending followup messages, assigning severities and tags, marking bugs as forwarded, and other issues.

Operations such as reassigning bugs to other packages, merging separate bug reports about the same issue, or reopening bugs when they are prematurely closed, are handled using the so-called control mail server. All of the commands available on this server are described in the BTS control server documentation.

5.9.1 Monitoring bugs

If you want to be a good maintainer, you should periodically check the Debian bug tracking system (BTS) for your packages. The BTS contains all the open bugs against your packages. You can check them by browsing this page: `https://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 https://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.

5.9.2 Responding to bugs

When responding to bugs, make sure that any discussion you have about bugs is sent to the original submitter of the bug, the bug itself and (if you are not the maintainer of the package) the maintainer. Sending an email to *123*`@bugs.debian.org` will send the mail to the maintainer of the package and record your email with the bug log. If you don't remember the submitter email address, you can use *123*`-submitter@bugs.debian.org` to also contact the submitter of the bug. The latter address also records the email with the bug log, so if you are the maintainer of the package in question, it is enough to send the reply to *123*`-submitter@bugs.debian.org`. Otherwise you should include *123*`@bugs.debian.org` so that you also reach the package maintainer.

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

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

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.

5.9.3 Bug housekeeping

As a package maintainer, you will often find bugs in other packages or have bugs reported against your packages which are actually bugs in other packages. The bug tracking system's features are described in the BTS documentation for Debian developers. Operations such as reassigning, merging, and tagging bug reports are described in the BTS control server documentation. 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* for details. However, handling the bugs in your own packages is even more important.

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

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

   If the bug submitter disagrees with your decision to close the bug, they may reopen it until you find an agreement on how to handle it. If you don't find any, you may want to tag the bug `wontfix` to let people know that the bug exists but that it won't be corrected. Please make sure that the bug submitter understands the reasons for your decision by adding an explanation to the message that adds the `wontfix` tag.

If this situation is unacceptable, you (or the submitter) may want to require a decision of the technical committee by filing a new bug against the `tech-ctte` pseudo-package with a summary of the situation. Before doing so, please read the recommended procedure.

2. If the bug is real but it's caused by another package, just reassign the bug to the right package. If you don't know which package it should be reassigned to, you should ask for help on *IRC channels* or on `debian-devel@lists.debian.org`. Please inform the maintainer(s) of the package you reassign the bug to, for example by Cc:ing the message that does the reassign to *packagename*`@packages.debian.org` and explaining your reasons in that mail. Please note that a simple reassignment is *not* e-mailed to the maintainers of the package being reassigned to, so they won't know about it until they look at a bug overview for their packages.

   If the bug affects the operation of your package, please consider cloning the bug and reassigning the clone to the package that really causes the behavior. Otherwise, the bug will not be shown in your package's bug list, possibly causing users to report the same bug over and over again. You should block "your" bug with the reassigned, cloned bug to document the relationship.

3. 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.

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

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

6. 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`. See *Coordination with upstream developers* if it's an upstream problem. If you have the required skills you can prepare a patch that fixes the bug and send it to the author at the same time. Make sure to send the patch to the BTS and to tag the bug as `patch`.

7. If you have fixed a bug in your local copy, or if a fix has been committed to the VCS 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.

8. Once a corrected package is available in the archive, the bug should be closed indicating the version in which it was fixed. This can be done automatically; read *When bugs are closed by new uploads*.

5.9.4 When bugs are closed by new uploads

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

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

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

  * Frobbed with options (closes: Bug#98339)
  * Added safety to prevent operator dismemberment, closes: bug#98765,
```

(continues on next page)

(continued from previous page)

```
    bug#98713, #98714.
  * Added man page. Closes: #98725.
```

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

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

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

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

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

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

For general information on how to write your changelog entries, see *Best practices for debian/changelog*.

5.9.5 Handling security-related bugs

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

When you become aware of a security-related bug in a Debian package, whether or not you are the maintainer, collect pertinent information about the problem, and promptly contact the security team by emailing `team@security.debian.org`. If desired, email can be encrypted with the Debian Security Contact key, see https://www.debian.org/security/faq#contact for details. **DO NOT UPLOAD** any packages for `stable` without contacting the team. Useful information includes, for example:

- Whether or not the bug is already public.
- Which versions of the package are known to be affected by the bug. Check each version that is present in a supported Debian release, as well as `testing` and `unstable`.
- The nature of the fix, if any is available (patches are especially helpful)
- Any fixed packages that you have prepared yourself (send the resulting debdiff or alternatively only the `.diff.gz` and `.dsc` files and read *Preparing packages to address security issues* first)
- Any assistance you can provide to help with testing (exploits, regression testing, etc.)
- Any information needed for the advisory (see *Security Advisories*)

As the maintainer of the package, you have the responsibility to maintain it, even in the stable release. You are in the best position to evaluate patches and test updated packages, so please see the sections below on how to prepare packages for the Security Team to handle.

5.9.5.1 Debian Security Tracker

The security team maintains a central database, the Debian Security Tracker. This contains all public information that is known about security issues: which packages and versions are affected or fixed, and thus whether stable, testing and/or unstable are vulnerable. Information that is still confidential is not added to the tracker.

You can search it for a specific issue, but also on package name. Look for your package to see which issues are still open. If you can, please provide more information about those issues, or help to address them in your package. Instructions are on the tracker web pages.

5.9.5.2 Confidentiality

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

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

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

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

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

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

Please note that if secrecy is needed you may not upload a fix to `unstable` (or anywhere else, such as a public VCS 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.

The Security Team has a PGP-key to enable encrypted communication about sensitive issues. See the Security Team FAQ for details.

5.9.5.3 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. 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 identifiers, and any other information useful in cross-referencing the vulnerability

5.9.5.4 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 (Application Program Interface) or ABI (Application Binary Interface), 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*).

Be sure to verify the following items:

- **Target the right distribution** in your `debian/changelog`: *codename*`-security` (e.g. `trixie-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. Add `closes:` statements for any **Debian bugs** filed. Always include an external reference, preferably a **CVE identifier**, so that it can be cross-referenced. However, if a CVE identifier has not yet been assigned, do not wait for it but continue the process. The identifier can be cross-referenced later.
- 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, or one that conflicts with a binNMU. The convention is to append `+deb`*X*`u1` (where *X* is the major release number), e.g. `1:2.4.3-4+deb13u1`, of course increasing 1 for any subsequent uploads.
- 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,bz2,xz}``** 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*) or setup a chroot (see *pbuilder* and *debootstrap*).

5.9.5.5 Uploading the fixed package

Do **NOT** upload a package to the security upload queue (on `*.security.upload.debian.org`) 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://ftp.security.upload.debian.org/pub/SecurityUploadQueue/`.

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

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

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

5.10 Subscribing to package updates

As a maintainer of a package, you get email notifications for various kinds of events (uploads, BTS messages etc). You can subscribe to receive such notifications also for packages you are not a maintainer of (or packages you maintain collaboratively), by mailing `control@tracker.debian.org` with `subscribe <pkgname>` either in the subject or the body of the email (see https://qa.pages.debian.net/distro-tracker/usage/messages.html#email-messages for details).

5.11 Moving, removing, renaming, orphaning, adopting, and reintroducing packages

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

5.11.1 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 for details). You must ensure that you include the `.orig.tar.{gz,bz2,xz}` in your upload (even if you are not uploading a new upstream version), or it will not appear in the new section together with the rest of the package. If your new section is valid, it will be moved automatically. If it does not, then contact the ftpmasters in order to understand what happened.

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

[1] See the Debian Policy Manual for guidelines on what section a package belongs in.

5.11.2 Removing packages

If for some reason you want to completely remove a package (say, if it is an old compatibility library which is no longer required), you need to file a bug against `ftp.debian.org` asking that the package be removed; as with all bugs, this bug should normally have normal severity. The bug title should be in the form `RM:` *package [architecture list]* `--` *reason*, where *package* is the package to be removed and *reason* is a short summary of the reason for the removal request. *[architecture list]* is optional and only needed if the removal request only applies to some architectures, not all. Note that the `reportbug` will create a title conforming to these rules when you use it to report a bug against the `ftp.debian.org` pseudo-package.

If you want to remove a package you maintain, you should note this in the bug title by prepending `ROM` (Request Of Maintainer). There are several other standard acronyms used in the reasoning for a package removal; see https://ftp-master.debian.org/removals.html for a complete list. That page also provides a convenient overview of pending removal requests.

Note that removals can only be done for the `unstable`, `experimental` and `stable` distributions. 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. (Removals from `testing` are possible though by filing a removal bug report against the `release.debian.org` pseudo-package. See *Removals from testing*.)

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

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

Usually you only ask for the removal of a package maintained by yourself. If you want to remove another package, you have to get the approval of its maintainer. Should the package be orphaned and thus have no maintainer, you should first discuss the removal request on `debian-qa@lists.debian.org`. If there is a consensus that the package should be removed, you should reassign and retitle the `O:` bug filed against the `wnpp` package instead of filing a new bug as removal request.

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

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

Once the package has been removed, the package's bugs should be handled. They should either be reassigned to another package in the case where the actual code has evolved into another package (e.g. `libfoo12` was removed because `libfoo13` supersedes it) or closed if the software is simply no longer part of Debian. When closing the bugs, to avoid marking the bugs as fixed in versions of the packages in previous Debian releases, they should be marked as fixed in the version `<most-recent-version-ever-in-Debian>+rm`.

5.11.2.1 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.[4] 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.

[4] Though, if a package still is in the upload queue and hasn't been moved to Incoming yet, it can be removed. (see *Uploading to ftp-master*)

5.11.3 Replacing or renaming packages

When the upstream maintainers for one of your packages chose to rename their software (or you made a mistake naming your package), you should follow a two-step process to rename it. In the first step, change the `debian/control` file to reflect the new name and to replace, provide and conflict with the obsolete package name (see the Debian Policy Manual for details). Please note that you should only add a `Provides` relation if all packages depending on the obsolete package name continue to work after the renaming. 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 (see *Removing packages*). 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` or `foo_1.00.orig.tar.bz2`. This restriction gives each file on the ftp site a unique name, which helps to ensure consistency across the mirror network.

5.11.4 Orphaning a package

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

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

More information is on the WNPP web pages.

5.11.5 Adopting a package

A list of packages in need of a new maintainer is available in the Work-Needing and Prospective Packages list (WNPP). 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 without assent of the current maintainer — that would be package hijacking. You can, of course, contact the current maintainer and ask them for permission to take over the package.

However, when a package has been neglected by the maintainer, you might be able to take over package maintainership by following the package salvaging process as described in *Package Salvaging*. If you have reason to believe a maintainer is no longer active at all, see *Dealing with inactive and/or unreachable maintainers*.

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 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 Debian Package Tracker* 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.

5.11.6 Reintroducing packages

Packages are often removed due to release-critical bugs, absent maintainers, too few users or poor quality in general. While the process of reintroduction is similar to the initial packaging process, you can avoid some pitfalls by doing some historical research first.

You should check why the package was removed in the first place. This information can be found in the removal item in the news section of the PTS page for the package or by browsing the log of removals. The removal bug will tell you why the package was removed and will give some indication of what you will need to work on in order to reintroduce the package. It may indicate that the best way forward is to switch to some other piece of software instead of reintroducing the package.

It may be appropriate to contact the former maintainers to find out if they are working on reintroducing the package, interested in co-maintaining the package or interested in sponsoring the package if needed.

You should do all the things required before introducing new packages (*New packages*).

You should base your work on the latest packaging available that is suitable. That might be the latest version from `unstable`, which will still be present in the snapshot archive.

The version control system used by the previous maintainer might contain useful changes, so it might be a good idea to have a look there. Check if the `control` file of the previous package contained any headers linking to the version control system for the package and if it still exists.

Package removals from `unstable` (not `testing`, `stable` or `oldstable`) trigger the closing of all bugs related to the package. You should look through all the closed bugs (including archived bugs) and unarchive and reopen any that were closed in a version ending in `+rm` and still apply. Any that no longer apply should be marked as fixed in the correct version if that is known.

Package removals from unstable also trigger marking the package as removed in the *Debian Security Tracker*. Debian members should mark removed issues as unfixed in the security tracker repository and all others should contact the security team to report reintroduced packages.

5.12 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, when a maintainer uploads a (portable) source package with binaries for the `i386` architecture, it will be built for each of the other architectures, amounting to 8 more builds.

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

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

1. Make sure that your `Build-Depends` and `Build-Depends-Indep` settings in `debian/control` are set properly. The best way to validate this is to use the `debootstrap` package to create an `unstable` chroot environment (see *debootstrap*). 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*).

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

   See the Debian Policy Manual 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. Setting your architecture to only one architecture (such as `i386` or `amd64`) 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 set up in a special way. Try building your package on another machine, even if it's the same architecture.
6. Don't depend on the package you're building being installed already (a sub-case of the above issue). There are, of course, exceptions to this rule, but be aware that any case like this needs manual bootstrapping and cannot be done by automated package builders.
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`.
9. When you can't support your package on a particular architecture, you shouldn't use the Architecture field to reflect that (it's also a pain to maintain correctly). If the package fails to build from source, you can just let it be and interested people can take a look at the build logs. If the package would actually build, the trick is to add a `Build-Depends` on `unsupported-architecture [the-not-supported-arch]`. The buildds will not build the package as the build dependencies are not fulfilled on that arch. To prevent building on 32-bits architectures, the `architecture-is-64-bit` build dependency can be used, as `architecture-is-little-endian` can be used to prevent building on big endian systems.

5.12.2 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 *When and how to do an NMU* instead.

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

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

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

5.12.2.1 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, etc.). 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 (`dak` refuses to install new packages if they don't have a version number greater than the currently available one).

You have to make sure that your binary-only NMU doesn't render the package uninstallable. This could happen when a source package generates arch-dependent and arch-independent packages that have inter-dependencies generated using dpkg's substitution variable `$(Source-Version)`.

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

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

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

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

5.12.2.2 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)*, just like non-porters. However, it is expected that the wait cycle for a porter's source NMU is smaller than for a non-porter, since porters have to cope with a large quantity of packages. Again, the situation varies depending on the distribution they are uploading to. It also varies whether the architecture is a candidate for inclusion into the next stable release; the release managers decide and announce which architectures are candidates.

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

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

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

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

5.12.3 Porting infrastructure and automation

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

5.12.3.1 Mailing lists and web pages

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

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

[2] In the past, such NMUs used the third-level number on the Debian part of the revision to denote their recompilation-only status; however, this syntax was ambiguous with native packages and did not allow proper ordering of recompile-only NMUs, source NMUs, and security NMUs on the same package, and has therefore been abandoned in favor of this new syntax.

5.12.3.2 Porter tools

Descriptions of several porting tools can be found in *Porting tools*.

5.12.3.3 `wanna-build`

The `wanna-build` system is used as a distributed, client-server build distribution system. It is usually used in conjunction with build daemons running the `buildd` program. `Build daemons` are `slave` hosts, which contact the central `wanna-build` system to receive a list of packages that need to be built.

`wanna-build` is not yet available as a package; however, all Debian porting efforts are using it for automated package building. The tool used to do the actual package builds, `sbuild`, is available as a package; see its description in *sbuild*. Please note that the packaged version is not the same as the one used on build daemons, but it is close enough to reproduce problems.

Most of the data produced by `wanna-build` that is generally useful to porters is available on the web at https://buildd.debian.org/. This data includes nightly updated statistics, queueing information and logs for build attempts.

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

The wanna-build team, in charge of the buildds, can be reached at `debian-wb-team@lists.debian.org`. To determine who (wanna-build team, release team) and how (mail, BTS) to contact, refer to https://lists.debian.org/debian-project/2009/03/msg00096.html.

When requesting binNMUs or give-backs (retries after a failed build), please use the format described at https://release.debian.org/wanna-build.txt.

5.12.4 When your package is *not* portable

Some packages still have issues with building and/or working on some of the architectures supported by Debian, and cannot be ported at all, or not within a reasonable amount of time. An example is a package that is SVGA-specific (only available for `i386` and `amd64`), or uses other hardware-specific features not supported on all architectures.

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

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

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

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

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

5.12.5 Marking non-free packages as auto-buildable

By default packages from the `non-free` and `non-free-firmware` sections are not built by the autobuilder network (mostly because the license of the packages could disapprove). To enable a package to be built, you need to perform the following steps:

1. Check whether it is legally allowed and technically possible to auto-build the package;
2. Add `XS-Autobuild: yes` into the header part of `debian/control`;

3. Send an email to `non-free@buildd.debian.org` and explain why the package can legitimately and technically be auto-built.

5.13 Non-Maintainer Uploads (NMUs)

Every package has one or more maintainers. Normally, these are the people who work on and upload new versions of the package. In some situations, it is useful that other developers can upload a new version as well, for example if they want to fix a bug in a package they don't maintain, when the maintainer needs help to respond to issues. Such uploads are called *Non-Maintainer Uploads (NMU)*.

5.13.1 When and how to do an NMU

Before doing an NMU, consider the following questions:

- Have you geared the NMU towards helping the maintainer? As there might be disagreement on the notion of whether the maintainer actually needs help or not, the DELAYED queue exists to give time to the maintainer to react and has the beneficial side-effect of allowing for independent reviews of the NMU diff.
- Does your NMU really fix bugs? ("Bugs" means any kind of bugs, e.g. wishlist bugs for packaging a new upstream version, but care should be taken to minimize the impact to the maintainer.) Using NMUs to make changes that are likely to be non-consensual is discouraged.
- As more specific examples, the following changes are generally considered acceptable, unless there are good reasons for not following those practices in a particular package: using the latest released debhelper compatibility level; using dh; using 3.0 (quilt); using lintian-brush.
- Did you give enough time to the maintainer? When was the bug reported to the BTS? Being busy for a week or two isn't unusual. Is the bug so severe that it needs to be fixed right now, or can it wait a few more days?
- How confident are you about your changes? Please remember the Hippocratic Oath: "Above all, do no harm." It is better to leave a package with an open grave bug than applying a non-functional patch, or one that hides the bug instead of resolving it. If you are not 100% sure of what you did, it might be a good idea to seek advice from others. Remember that if you break something in your NMU, many people will be very unhappy about it.
- Have you clearly expressed your intention to NMU, at least in the BTS? If that didn't generate any feedback, it might also be a good idea to try to contact the maintainer by other means (email to the maintainer addresses or private email, IRC).
- If the maintainer is usually active and responsive, have you tried to contact them? In general it should be considered preferable that maintainers take care of an issue themselves and that they are given the chance to review and correct your patch, because they can be expected to be more aware of potential issues which an NMUer might miss. It is often a better use of everyone's time if the maintainer is given an opportunity to upload a fix on their own.

When doing an NMU, you must first make sure that your intention to NMU is clear. Then, you must send a patch with the differences between the current package and your proposed NMU to the BTS. The `nmudiff` script in the `devscripts` package might be helpful.

While preparing the patch, you had better be aware of any package-specific practices that the maintainer might be using. Taking them into account reduces the burden of integrating your changes into the normal package workflow and thus increases the chances that integration will happen. A good place to look for possible package-specific practices is `debian/README.source`.

Unless you have an excellent reason not to do so, you must then give some time to the maintainer to react (for example, by uploading to the `DELAYED` queue). Here are some recommended values to use for delays:

- Upload fixing only release-critical bugs older than 7 days, with no maintainer activity on the bug for 7 days and no indication that a fix is in progress: 0 days
- Upload fixing only release-critical bugs older than 7 days: 2 days
- Upload fixing only release-critical and important bugs: 5 days
- Other NMUs: 15 days

Those delays are only examples. In some cases, such as uploads fixing security issues, or fixes for trivial bugs that block a transition, it is desirable that the fixed package reaches `unstable` sooner.

Sometimes, release managers decide to encourage NMUs with shorter delays for a subset of bugs (e.g release-critical bugs older than 7 days). Also, some maintainers list themselves in the Low Threshold NMU list, and accept that NMUs are uploaded without delay. But even in those cases, it's still a good idea to give the maintainer a few days to react before you upload, especially if the patch wasn't available in the BTS before, or if you know that the maintainer is generally active.

After you upload an NMU, you are responsible for the possible problems that you might have introduced. You must keep an eye on the package (*Subscribing to package updates* is a good way to achieve this).

This is not a license to perform NMUs thoughtlessly. If you NMU when it is clear that the maintainers are active and would have acknowledged a patch in a timely manner, or if you ignore the recommendations of this document, your upload might be a cause of conflict with the maintainer. You should always be prepared to defend the wisdom of any NMU you perform on its own merits.

5.13.2 NMUs and `debian/changelog`

Just like any other (source) upload, NMUs must add an entry to `debian/changelog`, telling what has changed with this upload. The first line of this entry must explicitly mention that this upload is an NMU, e.g.:

```
* Non-maintainer upload.
```

The way to version NMUs differs for native and non-native packages.

If the package is a native package (without a Debian revision in the version number), the version must be the version of the last maintainer upload, plus `+nmu`*X*, where *X* is a counter starting at `1`. If the last upload was also an NMU, the counter should be increased. For example, if the current version is `1.5`, then an NMU would get version `1.5+nmu1`.

If the package is not a native package, you should add a minor version number to the Debian revision part of the version number (the portion after the last hyphen). This extra number must start at `1`. For example, if the current version is `1.5-2`, then an NMU would get version `1.5-2.1`. If a new upstream version is packaged in the NMU, the Debian revision is set to `0`, for example `1.6-0.1`.

In both cases, if the last upload was also an NMU, the counter should be increased. For example, if the current version is `1.5+nmu3` (a native package which has already been NMUed), the NMU would get version `1.5+nmu4`.

A special versioning scheme is needed to avoid disrupting the maintainer's work, since using an integer for the Debian revision will potentially conflict with a maintainer upload already in preparation at the time of an NMU, or even one sitting in the ftp NEW queue. It also has the benefit of making it visually clear that a package in the archive was not made by the official maintainer.

If you upload a package to testing or stable, you sometimes need to "fork" the version number tree. This is the case for security uploads, for example. For this, a version of the form `+deb`*X*`u`*Y* should be used, where *X* is the major release number, and *Y* is a counter starting at `1`. For example, while `trixie` (Debian 13) is stable, a security NMU to stable for a package at version `1.5-3` would have version `1.5-3+deb13u1`, whereas a security upload to `forky` would get version `1.5-3+deb14u1`.

5.13.3 Using the `DELAYED/` queue

Having to wait for a response after you request permission to NMU is inefficient, because it costs the NMUer a context switch to come back to the issue. The `DELAYED` queue (see *Delayed uploads*) allows the developer doing the NMU to perform all the necessary tasks at the same time. For instance, instead of telling the maintainer that you will upload the updated package in 7 days, you should upload the package to `DELAYED/7` and tell the maintainer that they have 7 days to react. During this time, the maintainer can ask you to delay the upload some more, or cancel your upload.

You can cancel your upload using *dcut*. In case you uploaded `foo_1.2-1.1_all.changes` to a `DELAYED` queue, you can run `dcut cancel foo_1.2-1.1_all.changes` to cancel your upload. The `.changes` file does not need to be present locally as you instruct *dcut* to upload a command file removing a remote filename. The `.changes` file name is the same that you used when uploading.

The DELAYED queue should not be used to put additional pressure on the maintainer. In particular, it's important that you are available to cancel or delay the upload before the delay expires since the maintainer cannot cancel the upload themselves.

If you make an NMU to DELAYED and the maintainer updates the package before the delay expires, your upload will be rejected because a newer version is already available in the archive. Ideally, the maintainer will take care to include your proposed changes (or at least a solution for the problems they address) in that upload.

5.13.4 NMUs from the maintainer's point of view

When someone NMUs your package, this means they want to help you to keep it in good shape. This gives users fixed packages faster. You can consider asking the NMUer to become a co-maintainer of the package. Receiving an NMU on a package is not a bad thing; it just means that the package is interesting enough for other people to work on it.

To acknowledge an NMU, include its changes and changelog entry in your next maintainer upload. If you do not acknowledge the NMU by including the NMU changelog entry in your changelog, the bugs will remain closed in the BTS but will be listed as affecting your maintainer version of the package.

Note that if you ever need to revert a NMU that packages a new upstream version, it is recommended to use a fake upstream version like *CURRENT*+really*FORMER* until one can upload the latest version again. More information can be found in https://www.debian.org/doc/debian-policy/ch-controlfields.html#epochs-should-be-used-sparingly.

Note that easiest way to both check if your package has been NMUed, and also automatically download and commit the changes into a git-buildpackage maintained git repository is to run `gbp import-dsc --verbose --pristine-tar apt:<package>/sid`. This example command assumes you are working on the `debian/latest` branch preparing the next upload to Debian unstable, and it assumes your `apt` has the `deb-src` line active for Debian unstable.

5.13.5 Source NMUs vs Binary-only NMUs (binNMUs)

The full name of an NMU is *source NMU*. There is also another type, namely the *binary-only NMU*, or *binNMU*. A binNMU is also a package upload by someone other than the package's maintainer. However, it is a binary-only upload.

When a library (or other dependency) is updated, the packages using it may need to be rebuilt. Since no changes to the source are needed, the same source package is used.

BinNMUs are usually triggered on the buildds by wanna-build. An entry is added to `debian/changelog`, explaining why the upload was needed and increasing the version number as described in *Recompilation or binary-only NMU*. This entry should not be included in the next upload.

Buildds upload packages for their architecture to the archive as binary-only uploads. Strictly speaking, these are binNMUs. However, they are not normally called NMU, and they don't add an entry to `debian/changelog`.

5.13.6 NMUs vs QA uploads

NMUs are uploads of packages by somebody other than their assigned maintainer. There is another type of upload where the uploaded package is not yours: QA uploads. QA uploads are uploads of orphaned packages.

QA uploads are very much like normal maintainer uploads: they may fix anything, even minor issues; the version numbering is normal, and there is no need to use a delayed upload. The difference is that you are not listed as the `Maintainer` or `Uploader` for the package. Also, the changelog entry of a QA upload has a special first line:

```
* QA upload.
```

If you want to do an NMU, and it seems that the maintainer is not active, it is wise to check if the package is orphaned (this information is displayed on the package's Package Tracking System page). When doing the first QA upload to an orphaned package, the maintainer should be set to `Debian QA Group <packages@qa.debian.org>`. Orphaned packages which did not yet have a QA upload still have their old maintainer set. There is a list of them at https://qa.debian.org/orphaned.html.

Instead of doing a QA upload, you can also consider adopting the package by making yourself the maintainer. You don't need permission from anybody to adopt an orphaned package; you can just set yourself as maintainer and upload the new version (see *Adopting a package*).

5.13.7 NMUs vs team uploads

Sometimes you are fixing and/or updating a package because you are member of a packaging team (which uses a mailing list as `Maintainer` or `Uploader`; see *Collaborative maintenance*) but you don't want to add yourself to `Uploaders` because you do not plan to contribute regularly to this specific package. If it conforms with your team's policy, you can perform a normal upload without being listed directly as `Maintainer` or `Uploader`. In that case, you should start your changelog entry with the following line:

```
* Team upload.
```

5.14 Package Salvaging

Package salvaging is the process by which one attempts to save a package that, while not officially orphaned, appears poorly maintained or completely unmaintained. This is a weaker and faster procedure than orphaning a package officially through the powers of the MIA team. Salvaging a package is not meant to replace MIA handling, and differs in that it does not imply anything about the overall activity of a maintainer. Instead, it handles a package maintainership transition for a single package only, leaving any other package or Debian membership or upload rights (when applicable) untouched.

Note that the process is only intended for actively taking over maintainership. Do not start a package salvaging process when you do not intend to maintain the package for a prolonged time. If you only want to fix certain things, but not take over the package, you must use the NMU process, even if the package would be eligible for salvaging. The NMU process is explained in *Non-Maintainer Uploads (NMUs)*.

Another important thing to remember: It is not acceptable to hijack others' packages. If followed, this salvaging process will help you to ensure that your endeavour is not a hijack but a (legal) salvaging procedure, and you can counter any allegations of hijacking with a reference to this process. Thanks to this process, new contributors should no longer be afraid to take over packages that have been neglected or entirely forgotten.

The process is split into two phases: In the first phase you determine whether the package in question is *eligible* for the salvaging process. Only when the eligibility has been determined you may enter the second phase, the *actual* package salvaging.

For additional information, rationales and FAQs on package salvaging, please visit the Salvaging Packages page on the Debian wiki.

5.14.1 When a package is eligible for package salvaging

A package becomes eligible for salvaging when it has been neglected by the current maintainer. To determine that a package has really been neglected by the maintainer, the following indicators give a rough idea what to look for:

- NMUs, especially if there has been more than one NMU in a row.
- Bugs filed against the package do not have answers from the maintainer.
- Upstream has released several versions, but despite there being a bug entry asking for it, it has not been packaged.
- There are QA issues with the package.

You will have to use your judgement as to whether a given combination factors constitutes neglect; in case the maintainer disagrees they have only to say so (see below). If you're not sure about your judgement or simply want to be on the safe side, there is a more precise (and conservative) set of conditions in the Package Salvaging wiki page. These conditions represent a current Debian consensus on salvaging criteria. In any case you should explain your reasons for thinking the package is neglected when you file an Intent to Salvage bug later.

5.14.2 How to salvage a package

If and *only* if a package has been determined to be eligible for package salvaging, any prospective maintainer may start the following package salvaging procedure.

1. Open a bug with the severity "important" against the package in question, expressing the intent to take over maintainership of the package. For this, the title of the bug should start with `ITS: package-name`[3]. You may alternatively offer to only take co-maintenance of the package. When you file the bug, you must inform all maintainers, uploaders and if applicable the packaging team explicitly by adding them to `X-Debbugs-CC`. Additionally, if the maintainer(s) seem(s) to be generally inactive, please inform the MIA team by adding `mia@qa.debian.org` to `X-Debbugs-CC` as well. As well as the explicit expression of the intent to salvage, please also take the time to document your assessment of the eligibility in the bug report, for example by listing the criteria you've applied and adding some data to make it easier for others to assess the situation.

2. In this step you need to wait in case any objections to the salvaging are raised; the maintainer, any current uploader or any member of the associated packaging team of the package in question may object publicly in response to the bug you've filed within `21 days`, and this terminates the salvaging process.

   The current maintainers may also agree to your intent to salvage by filing a (signed) public response to the the bug. They might propose that you become a co-maintainer instead of the sole maintainer. On team maintained packages, a member of the associated team can accept your salvaging proposal by sending out a signed agreement notice to the ITS bug, alternatively inviting you to become a new co-maintainer of the package. The team may require you to keep the package under the team's umbrella, but then may ask or invite you to join the team. In any of these cases where you have received the OK to proceed, you can upload the new package immediately as the new (co-)maintainer, without the need to utilise the `DELAYED` queue as described in the next step.

3. After the 21 days delay, if no answer has been sent to the bug from the maintainer, one of the uploaders or team, you may upload the new release of the package into the `DELAYED` queue with a minimum delay of `seven days`. You should close the salvage bug in the changelog and you must also send an nmudiff to the bug ensuring that copies are sent to the maintainer and any uploaders (including teams) of the package by `CC'ing` them in the mail to the BTS.

   During the waiting time of the `DELAYED` queue, the maintainer can accept the salvaging, do an upload themselves or (ask to) cancel the upload. The latter two of these will also stop the salvaging process, but the maintainer must reply to the salvaging bug with more information about their action.

5.15 Collaborative maintenance

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

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

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

- Set up 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 `Git`. Salsa (see *salsa.debian.org: Git repositories and collaborative development platform*) provides Git repositories, amongst other collaborative tools.
- Add the co-maintainer's correct maintainer name and address to the `Uploaders` field in the first paragraph of the `debian/control` file.

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

- The co-maintainers should subscribe themselves to the appropriate source package (see *Subscribing to package updates*).

[3] ITS is shorthand for *"Intend to Salvage"*

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

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

In any case, it is a bad idea to automatically put all team members in the `Uploaders` field. It clutters the Developer's Package Overview listing (see *Developer's packages overview*) with packages one doesn't really care for, and creates a false sense of good maintenance. For the same reason, team members do not need to add themselves to the `Uploaders` field just because they are uploading the package once, they can do a "Team upload" (see *NMUs vs team uploads*). Conversely, it is a bad idea to keep a package with only the mailing list address as a `Maintainer` and no `Uploaders`.

5.16 The testing distribution

5.16.1 Basics

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

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

5.16.2 Updates from unstable

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

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

- The package must have been available in `unstable` for a certain number of days, see *Selecting the upload urgency*. Please note that the urgency is sticky, meaning that the highest urgency uploaded since the previous `testing` transition is taken into account;
- It must not have new release-critical bugs (RC bugs affecting the version available in `unstable`, but not affecting the version in `testing`);
- It must be available on all architectures on which it has previously been built in `unstable`. *The dak ls utility* may be of interest to check that information;
- It must not break any dependency of a package which is already available in `testing`;
- The packages on which it depends must either be available in `testing` or they must be accepted into `testing` at the same time (and they will be if they fulfill all the necessary criteria);
- The phase of the project. I.e. automatic transitions are turned off during the *freeze* of the `testing` distribution.

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, or use the program `grep-excuses` which is in the `devscripts` package. This utility can easily be used in a crontab 5 to keep yourself informed of the progression of your packages into `testing`.

The `update_excuses` file does not always give the precise reason why the package is refused; you may have to find it on your own by looking for what would break with the inclusion of the package. The testing web page gives some more information about the usual problems which may be causing such troubles.

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

Some further dependency analysis is shown on https://release.debian.org/migration/ — but be warned: this page also shows build dependencies that are not considered by britney.

5.16.2.1 Out-of-date

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

Consider this example:

	alpha	arm
testing	1	-
unstable	1	2

The package is out of date on `alpha` in `unstable`, and will not go to `testing`. Removing the package would not help at all; the package is still out of date on `alpha`, and will not propagate to `testing`.

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

	alpha	arm	hurd-i386
testing	1	1	-
unstable	2	-	1

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

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

5.16.2.2 Removals from testing

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

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

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

5.16.2.3 Circular dependencies

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

An example of this is:

	testing	unstable
a	1; depends: b=1	2; depends: b=2
b	1; depends: a=1	2; depends: a=2

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

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

5.16.2.4 Influence of package in testing

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

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

5.16.2.5 Details

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

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

Now, the more complex part happens: Britney tries to update `testing` with the valid candidates. For that, britney tries to add each valid candidate to the testing distribution. If the number of uninstallable packages in `testing` doesn't increase, the package is accepted. From that point on, the accepted package is considered to be part of `testing`, such that all subsequent installability tests include this package. Hints from the release team are processed before or after this main run, depending on the exact type.

If you want to see more details, you can look it up on https://release.debian.org/britney/update_output/.

The hints are available via https://release.debian.org/britney/hints/, where you can find the description as well. With the hints, the Debian Release team can block or unblock packages, ease or force packages into `testing`, remove packages from `testing`, approve uploads to *Direct updates to testing* or override the urgency.

5.16.3 Direct updates to testing

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

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

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

Version numbers are usually selected by appending `+debXuY`, where *X* is the major release number of Debian and *Y* is a counter starting at 1. e.g. `1:2.4.3-4+deb13u1`.

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

- Make sure that your package really needs to go through `testing-proposed-updates`, and can't go through `unstable`;
- Make sure that you included only the minimal amount of changes;
- Make sure that you included an appropriate explanation in the changelog;
- Make sure that you've written the testing *Release code names* (e.g. `forky`) into your target distribution;
- Make sure that you've built and tested your package in `testing`, not in `unstable`;
- Make sure that your version number is higher than the version in `testing` and `testing-proposed-updates`, and lower than in `unstable`;

- Ask for authorization for uploading from the release managers.
- After uploading and successful build on all platforms, contact the release team at `debian-release@lists.debian.org` and ask them to approve your upload.

5.16.4 Frequently asked questions

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

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

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

The `unstable` bug count comprises all release-critical bugs that are marked to apply to *package/version* combinations available in `unstable` for a release architecture. The `testing` bug count is defined analogously.

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

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

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

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

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

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

5.17 The Stable backports archive

5.17.1 Basics

Once a package reaches the `testing` distribution, it is possible for anyone with upload rights in Debian (see below about this) to build and upload a backport of that package to `stable-backports`, to allow easy installation of the version from `testing` onto a system that is tracking the `stable` distribution.

One should not upload a version of a package to `stable-backports` until the matching version has already reached the `testing` archive.

5.17.2 Exception to the testing-first rule

The only exception to the above rule, is when there's an important security fix that deserves a quick upload: in such a case, there is no need to delay an upload of the security fix to the `stable-backports` archive. However, it is strongly advised that the package is first fixed in `unstable` before uploading a fix to the `stable-backports` archive.

5.17.3 Who can maintain packages in the stable-backports archive?

It is not necessarily up to the original package maintainer to maintain the `stable-backports` version of the package. Anyone can do it, and one doesn't even need approval from the original maintainer to do so. It is however good practice to first get in touch with the original maintainer of the package before attempting to start the maintenance of a package in `stable-backports`. The maintainer can, if they wish, decide to maintain the backport themselves, or help you doing so. It is not uncommon, for example, to apply a patch to the unstable version of a package, to facilitate its backporting.

5.17.4 When can one start uploading to stable-backports?

The new `stable-backports` is created before the freeze of the next `stable` suite. However, it is not allowed to upload there until the very end of the freeze cycle. The `stable-backports` archive is usually opened a few weeks before the final release of the next `stable` suite, but it doesn't make sense to upload until the release has actually happened.

5.17.5 How long must a package be maintained when uploaded to stable-backports?

The `stable-backports` archive is maintained for bugs and security issues during the whole life-cycle of the Debian `stable` suite. Therefore, an upload to `stable-backports`, implies a willingness to maintain the backported package for the duration of the `stable` suite, which can be expected to be about 3 years from its initial release.

The person uploading to backports is also supposed to maintain the backported packages for security during the lifetime of `stable`.

It is to be noted that the `stable-backports` isn't part of the LTS or ELTS effort. The `stable-backports` FTP masters will close the `stable-backports` repositories for uploads once `stable` reaches end-of-life (ie: when `stable` becomes maintained by the LTS team only). Therefore there won't be any maintenance of packages from `stable-backports` after the official end of life of the `stable` suite, as uploads will not be accepted.

5.17.6 How often shall one upload to stable-backports?

The packages in backports are supposed to follow the developments that are happening in Testing. Therefore, it is expected that any significant update in `testing` should trigger an upload into `stable-backports`, until the new `stable` is released. However, please do not backport minor version changes without user visible changes or bugfixes.

5.17.7 How can one learn more about backporting?

You can learn more about how to contribute directly on the backport web site.

It is also recommended to read the Frequently Asked Questions (FAQ).

CHAPTER

SIX

BEST PACKAGING PRACTICES

Debian's quality is largely due to the Debian Policy, which defines explicit baseline requirements that 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 that go into the package (directly or indirectly), it's usually the file maintainers spend the most time on.

6.1.1 Helper scripts

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

Helper scripts take care of these issues. Assuming you comply with the conventions expected by the helper script, the helper takes care of all the details. Changes in policy can be made in the helper script; then packages just need to be rebuilt with the new version of the helper and no other changes.

Overview of Debian Maintainer Tools 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(7) and looking at the examples that come with the package. `dh_make`, from the `dh-make` package (see *dh-make*), 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.

6.1.2 Multiple binary packages

A single source package will often build several binary packages, either to provide several flavors of the same software (e.g., the `vim` source package) or to make several small packages instead of a big one (e.g., so the user can install only the subset needed, and thus save some disk space, see for example the `libxml2` source package).

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 a 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.

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* describes common guidelines for both parts of the package description. Following that, *The package synopsis, or short description* provides guidelines specific to the synopsis, and *The long description* contains guidelines specific to the description.

6.2.1 The package name

The package name:

- Must be consistent with widely established conventions, e.g.
  - C and C++ libraries are typically prefixed with lib
  - for many other languages, package names are prefixed or a suffixed with the language name (the actual convention depends on the language).
- Should not be a common, unqualified word. In particular, words that represent a whole class of applications (e.g. reader, browser) should be reserved for virtual packages.
- Should ideally be at least 4 characters long, unless the upstream name is shorter than that *and* the project is already widely recognized by that name (e.g. fzf).

6.2.2 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.

6.2.3 The package synopsis, or short description

Policy says the synopsis line (the short description) must be concise, not repeating the package name, but also informative.

The synopsis functions as a phrase describing the package, not a complete sentence, so sentential punctuation is inappropriate: it does not need extra capital letters or a final period (full stop). It should also omit any initial indefinite or definite article — "a", "an", or "the". Thus for instance:

```
Package: libeg0
Description: exemplification support library
```

Technically this is a noun phrase minus articles, as opposed to a verb phrase. A good heuristic is that it should be possible to substitute the package *name* and *synopsis* into this formula:

The package *name* provides {a,an,the,some} *synopsis*.

Sets of related packages may use an alternative scheme that divides the synopsis into two parts, the first a description of the whole suite and the second a summary of the package's role within it:

```
Package: eg-tools
Description: simple exemplification system (utilities)

Package: eg-doc
Description: simple exemplification system - documentation
```

These synopses follow a modified formula. Where a package "*name*" has a synopsis "*suite* (*role*)" or "*suite* - *role*", the elements should be phrased so that they fit into the formula:

The package *name* provides {a,an,the} *role* for the *suite*.

6.2.4 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.

Long descriptions of related packages, for example built from the same source, can share paragraphs in order to increase consistency and reduce the workload for translators, but you need at least one separate paragraph describing the package's specific role.

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., is this the client for the foo server)?

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

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

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

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

- What does the package do? If it is an add-on to another package, then the short description of the package we are an add-on to should be put in here.

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

6.2.5 Upstream home page

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

6.2.6 Version Control System location

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

6.2.6.1 Vcs-Browser

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

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

6.2.6.2 Vcs-*

Value of this field should be a string identifying unequivocally the location of the Version Control System repository used to maintain the given package, if available. * identifies the Version Control System; currently the following systems are supported by the package tracking system: `arch`, `bzr` (Bazaar), `cvs`, `darcs`, `git`, `hg` (Mercurial), `mtn` (Monotone), `svn` (Subversion).

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

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

```
Source: vim
<snip>
Vcs-Git: https://salsa.debian.org/vim-team/vim.git
Vcs-Browser: https://salsa.debian.org/vim-team/vim
Homepage: https://www.vim.org
```

Maintaining the packaging in a version control system, and setting a Vcs-* header is good practice and makes it easier for others to contribute changes.

Almost all packages in Debian that use a version control system use Git; if you create a new package, using Git is a good idea simply because it's the system that contributors will be familiar with.

DEP-14 defines a common layout for Debian packages.

6.3 Best practices for `debian/changelog`

The following practices supplement the Policy on changelog files.

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* for more information.

6.3.2 Selecting the upload urgency

The release team have indicated that they expect most uploads to `unstable` to use **urgency=medium**. That is, you should choose **urgency=medium** unless there is some particular reason for the upload to migrate to `testing` more quickly or slowly (see also *Updates from unstable*). For example, you might select **urgency=low** if the changes since the last upload are large and might be disruptive in unanticipated ways.

The delays are currently 2, 5 or 10 days, depending on the urgency (high, medium or low). The actual numbers are actually controled by the britney configuration which also includes accelerated migrations when Autopkgtest passes.

6.3.3 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*.

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* for more information on how to use the bug tracking system.

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

6.3.4 Common errors in changelog entries

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

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

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

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

What was the patch about?

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

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

```
* Fix vsync fw glitch w/ ancient CRTs.
```

Too many acronyms (what does "fw" mean, "firmware"?), and it's not overly clear what the glitch was actually about, or how it was fixed.

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

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

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

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

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

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

6.3.5 Supplementing changelogs with `NEWS.Debian` files

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

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

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

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

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

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

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

6.4 Best practices around security

When an upstream publishes a cryptographic signature for every new release, you should setup uscan to automatically verify the latter. For details, refer to the section on Upstream source location: "debian/watch" in the The Debian Policy Manual.

https://wiki.debian.org/Hardening has suggestions on how to build security hardened executables.

6.5 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 that isn't handled merely by the creation or removal of files and directories. The following instructions supplement the Debian Policy.

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

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

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

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

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

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

```
if command -v install-docs > /dev/null; then ...
```

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 best way, since command -v is a shell-builtin for many shells and is defined in POSIX.

Using which is an acceptable alternative, since it is from the required debianutils package.

6.6 Configuration management with debconf

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

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

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

6.6.1 Do not abuse debconf

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

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

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

6.6.2 General recommendations for authors and translators

6.6.2.1 Write correct English

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

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

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

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

6.6.2.2 Be kind to translators

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

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

Avoid changing templates too often. Changing template text induces more work for translators, which will get their translation fuzzied. A fuzzy translation is a string for which the original changed since it was translated, therefore requiring some update by a translator to be usable. When changes are small enough, the original translation is kept in PO files but marked as `fuzzy`.

If you plan to do changes to your original templates, please use the notification system provided with the `po-debconf` package, namely the `podebconf-report-po`, to contact translators. Most active translators are very responsive and getting their work included along with your modified templates will save you additional uploads. If you use gettext-based templates, the translator's name and e-mail addresses are mentioned in the PO files headers and will be used by `podebconf-report-po`.

A recommended use of that utility is:

```
cd debian/po && podebconf-report-po --call --languageteam --withtranslators --
↪deadline="+10 days"
```

This command will first synchronize the PO and POT files in `debian/po` with the template files listed in `debian/po/POTFILES.in`. Then, it will send a call for new translations, in the `debian-i18n@lists.debian.org` mailing list. Finally, it will also send a call for translation updates to the language team (mentioned in the `Language-Team` field of each PO file) as well as the last translator (mentioned in `Last-translator`).

Giving a deadline to translators is always appreciated, so that they can organize their work. Please remember that some translation teams have a formalized translate/review process and a delay lower than 10 days is considered as unreasonable. A shorter delay puts too much pressure on translation teams and should be kept for very minor changes.

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

6.6.2.3 Unfuzzy complete translations when correcting typos and spelling

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

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

To *unfuzzy* translations, you can use `msguntypot` (part of the `po4a` package).

1. Regenerate the POT and PO files.

```
debconf-updatepo
```

2. Make a copy of the POT file.

```
cp templates.pot templates.pot.orig
```

3. Make a copy of all the PO files.

```
mkdir po_fridge; cp *.po po_fridge
```

4. Change the debconf template files to fix the typos.
5. Regenerate the POT and PO files (again).

```
debconf-updatepo
```

   At this point, the typo fix fuzzied all the translations, and this unfortunate change is the only one between the PO files of your main directory and the one from the fridge. Here is how to solve this.

6. Discard fuzzy translation, restore the ones from the fridge.

```
cp po_fridge/*.po .
```

7. Manually merge the PO files with the new POT file, but taking the useless fuzzy into account.

```
msguntypot -o templates.pot.orig -n templates.pot *.po
```

8. Clean up.

```
rm -rf templates.pot.orig po_fridge
```

6.6.2.4 Do not make assumptions about interfaces

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

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

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

6.6.2.5 Do not use first person

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

6.6.2.6 Be gender neutral

As a way of showing our commitment to our diversity statement, please use gender-neutral constructions in your writing. This means avoiding pronouns like he/she when referring to a role (like "maintainer") whose gender is unknown. Instead, you should use the plural form (singular they).

6.6.3 Templates fields definition

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

6.6.3.1 Type

string

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

password

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

boolean

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

select

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

If choices are translatable strings, the 'Choices' field may be marked as translatable by using `__Choices`. The double underscore will split out each choice in a separate string.

The `po-debconf` system also offers interesting possibilities to only mark **some** choices as translatable. Example:

```
Template: foo/bar
Type: Select
#flag:translate:3
__Choices: PAL, SECAM, Other
_Description: TV standard:
 Please choose the TV standard used in your country.
```

In that example, only the 'Other' string is translatable while others are acronyms that should not be translated. The above allows only 'Other' to be included in PO and POT files.

The debconf templates flag system offers many such possibilities. The po-debconf(7) manual page lists all these possibilities.

multiselect

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

note

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

text

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

error

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

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

6.6.3.2 Description: short and extended description

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

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

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

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

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

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

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

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

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

6.6.3.3 Choices

This field should be used for select and multiselect types. It contains the possible choices that will be presented to users. These choices should be separated by commas.

6.6.3.4 Default

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

6.6.4 Template fields specific style guide

6.6.4.1 Type field

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

6.6.4.2 Description field

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

String/password templates

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

Boolean templates

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

Select/Multiselect

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

Notes

- The short description should be considered to be a **title**.
- The extended description is what will be displayed as a more detailed explanation of the note. Phrases, no terse writing style.
- **Do not abuse debconf.** Notes are the most common way to abuse debconf. As written in the debconf-devel(7) manual page: it's best to use them only for warning about very serious problems. The `NEWS.Debian` or `README.Debian` files are the appropriate location for a lot of notes. If, by reading this, you consider converting your Note type templates to entries in `NEWS.Debian` or `README.Debian`, please consider keeping existing translations for the future.

6.6.4.3 Choices field

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

6.6.4.4 Default field

If the default value for a select template is likely to vary depending on the user language (for instance, if the choice is a language choice), please use the _Default trick, documented in po-debconf 7.

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

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

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

Example, taken from the geneweb package templates:

```
Template: geneweb/lang
Type: select
__Choices: Afrikaans (af), Bulgarian (bg), Catalan (ca), Chinese (zh), Czech (cs),␣
↪Danish (da), Dutch (nl), English (en), Esperanto (eo), Estonian (et), Finnish (fi),␣
↪French (fr), German (de), Hebrew (he), Icelandic (is), Italian (it), Latvian (lv),␣
↪Norwegian (no), Polish (pl), Portuguese (pt), Romanian (ro), Russian (ru), Spanish␣
↪(es), Swedish (sv)
# This is the default choice. Translators may put their own language here
# instead of the default.
# WARNING : you MUST use the ENGLISH NAME of your language
# For instance, the French translator will need to put French (fr) here.
_Default: English[ translators, please see comment in PO files]
_Description: Geneweb default language:
```

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

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

6.7 Internationalization

This section contains global information for developers to make translators' lives easier. More information for translators and developers interested in internationalization are available in the Internationalisation and localisation in Debian documentation.

6.7.1 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 package 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.

6.7.2 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 is 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 in the `debian-installer` 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 VCS 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 that is included by all the different translations. This makes it much easier, for instance, to keep URLs up to date across multiple files.

Some tools (e.g. `po4a`, `poxml`, or the `translate-toolkit`) are specialized in extracting the translatable material from different formats. They produce PO files, a format quite common to translators, which permits seeing what needs to be re-translated when the translated document is updated.

6.8 Best practices for `debian/patches`

Debian packages might suffer from bugs in the upstream code that you need to deal with. In the source format "3.0 (quilt)" patches are stored in `debian/patches/` and automatically applied as listed in `debian/patches/series` when the source package is unpacked.

Patches should be documented following DEP-3.

Several tools exist to automate managing the patches. If you manage a source package outside of any Git repository, then your best option is likely `quilt`. Otherwise, you should consider to rely on Git's built-in features or on on the git packaging helper that you use (if any). In particular, for packages using `git-buildpackage`, you should use the `gbp pq` commands to manage the contents of the `debian/patches/` directory.

A single patch can be created with e.g. `git format-patch -1 d33286c` from a single commit. Avoid using *git show* as it lacks the full headers.

If the upstream fix is spread across multiple commits but makes sense to apply (and drop) in Debian as a single patch, one could use a command such as `git format-patch --stdout abc123..def456 > debian/patches/...` and append the `Bug` field only in the commit message of the first commit in the patch.

If one appends *.patch* to the url of a GitHub commit or Pull Request or GitLab commit or Merge Request, the resulting patch file is using this same format (as if it were generated by `git format-patch`).

Remember to always append a `Bug` header to the patch description so that a reader can follow the link to see where the bug was reported or patch submitted. If the purpose of the patch is to specifically divert from upstream permanently, append the header *Forwarded: not-needed* to the end of the description.

6.9 Common packaging situations

6.9.1 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.

6.9.2 Libraries

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

Good practices for library packaging have been grouped in the library packaging guide.

6.9.3 Documentation

Be sure to follow the Policy on documentation.

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

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

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

Debian policy (section 12.1) directs that manual pages should accompany every program, utility, and function, and suggests them for other objects like configuration files. If the work you are packaging does not have such manual pages, consider writing them for inclusion in your package, and submitting them upstream.

The manpages do not need to be written directly in the troff format. Popular source formats are DocBook, POD and reST, which can be converted using `xsltproc`, `pod2man` and `rst2man` respectively. To a lesser extent, the `help2man` program can also be used to write a stub.

6.9.4 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; 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.
- Java related packages have their 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`.
- Rust packaging is described in the Debian Rust Team Book;.
- Packages providing services ("daemons") should be functional on a fresh install, to the extent that that is possible without compromising security (e.g. a web server should by default be up and running and serve a dummy page, but must otherwise not allow unauthenticated sensitive operations; consider whether to serve only on the localhost network interface, by default).
- Web application packages should aim to have their dependencies (including javascript) packaged separately, and should carry out whatever setup is necessary for basic and secure functionality out of the box (e.g. create a database, ship configs with reasonable defaults, install files in appropriate location with appropriate permissions, etc). For examples, look at how existing web applications are packaged, e.g. dfsg-new-queue for Go, gitlab for ruby on rails, node-shiny-server for NPM. diaspora-installer is a dummy package which downloads diaspora (also pulling in runtime dependencies as rubygems) and configures it to use PostgreSQL and Nginx.

6.9.5 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 ten or more .debs, one per each architecture. While this adds some extra overhead into the `Packages` files, it saves a lot of disk space on Debian mirrors. Separating out architecture-independent data also reduces processing time of `lintian` (see *Package lint tools*) when run over the entire Debian archive.

6.9.6 Needing a certain locale during build

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

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

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

mkdir -p $LOCALE_PATH
localedef -i $LOCALE_NAME.$LOCALE_CHARSET -f $LOCALE_CHARSET $LOCALE_PATH/$LOCALE_
↪NAME.$LOCALE_CHARSET

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

6.9.7 Make transition packages deborphan compliant

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

For example, with `--guess-dummy`, `deborphan` tries to search all transitional packages which were needed for upgrade but which can now be removed. For that, it currently looks for the string dummy or transitional in their short description, though it would be better to search for both strings, as there are some dummy or transitional packages, which have other purposes.

So, when you are creating such a package, please make sure to add `transitional dummy package` to the short description to make this explicit. If you are looking for examples, just run: `apt-cache search .|grep dummy` or `apt-cache search .|grep transitional`.

Also, it is recommended to adjust its section to `oldlibs` and its priority to `optional` in order to ease `deborphan`'s job.

6.9.8 Best practices for `.orig.tar.{gz,bz2,xz}` files

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

6.9.8.1 Pristine source

The defining characteristic of a pristine source tarball is that the `.orig.tar.{gz,bz2,xz}` file is byte-for-byte identical to a tarball officially distributed by the upstream author.[1] This makes it possible to use checksums to easily verify that all changes between Debian's version and upstream's are contained in the Debian diff. Also, if the original source is huge, upstream authors and others who already have the upstream tarball can save download time if they want to inspect your packaging in detail.

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

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

   ```
   zcat path/to/packagename_upstream-version.orig.tar.gz | tar xf -
   ```

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

[1] We cannot prevent upstream authors from changing the tarball they distribute without also incrementing the version number, so there can be no guarantee that a pristine tarball is identical to what upstream *currently* distributing at any point in time. All that can be expected is that it is identical to something that upstream once *did* distribute. If a difference arises later (say, if upstream notices that they weren't using maximal compression in their original distribution and then re-`gzip` it), that's just too bad. Since there is no good way to upload a new `.orig.tar.{gz,bz2,xz}` for the same version, there is not even any point in treating this situation as a bug.

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

6.9.8.2 Repackaged upstream source

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

In these cases the developer must construct a suitable `.orig.tar.{gz,bz2,xz}` file themselves. We refer to such a tarball as a repackaged upstream source. Note that a repackaged upstream source is different from a Debian-native package. A repackaged source still comes with Debian-specific changes in a separate `.diff.gz` or `.debian.tar.{gz,bz2,xz}` and still has a version number composed of *upstream-version* and *debian-version*.

There may be cases where it is desirable to repackage the source even though upstream distributes a `.tar.{gz,bz2,xz}` that could in principle be used in its pristine form. The most obvious is if *significant* space savings can be achieved by recompressing the tar archive or by removing genuinely useless cruft from the upstream archive. Use your own discretion here, but be prepared to defend your decision if you repackage source that could have been pristine.

A repackaged `.orig.tar.{gz,bz2,xz}`

1. **should** be documented in the resulting source package. Detailed information on how the repackaged source was obtained, and on how this can be reproduced should be provided in `debian/copyright`, ideally in a way that can be done automatically with uscan. If that really doesn't work, at least provide a `get-orig-source` target in your `debian/rules` file that repeats the process, even though that was actually deprecated in the 4.1.4 version of the Debian policy.
2. **should not** contain any file that does not come from the upstream author(s), or whose contents has been changed by you.[2]
3. **should**, except where impossible for legal reasons, preserve the entire building and portability infrastructure provided by the upstream author. For example, it is not a sufficient reason for omitting a file that it is used only when building on MS-DOS. Similarly, a `Makefile` provided by upstream should not be omitted even if the first thing your `debian/rules` does is to overwrite it by running a configure script.

   (*Rationale:* It is common for Debian users who need to build software for non-Debian platforms to fetch the source from a Debian mirror rather than trying to locate a canonical upstream distribution point).
4. **may** use *packagename-upstream-version*`+dfsg` (or any other suffix which is added to the tarball name) as the name of the top-level directory in its tarball. This makes it possible to distinguish pristine tarballs from repackaged ones.
5. **should** be compressed with `xz` (or `gzip` or `bzip`) with maximal compression.

6.9.8.3 Changing binary files

Sometimes it is necessary to change binary files contained in the original tarball, or to add binary files that are not in it. This is fully supported when using source packages in "3.0 (quilt)" format; see the dpkg-source(1) manual page for details. When using the older format "1.0", binary files can't be stored in the `.diff.gz` so you must store a `uuencoded` (or similar) version of the file(s) and decode it at build time in `debian/rules` (and move it in its official location).

6.9.9 Best practices for debug packages

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

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

The debug packages contain separated debugging symbols that gdb can find and load on the fly when debugging a program or library. The convention in Debian is to keep these symbols in /usr/lib/debug/*path*, where *path* is the path to the executable or library. For example, debugging symbols for /usr/bin/foo go in /usr/lib/debug/usr/bin/foo, and debugging symbols for /usr/lib/libfoo.so.1 go in /usr/lib/debug/usr/lib/libfoo.so.1.

6.9.9.1 Automatically generated debug packages

Debug symbol packages can be generated automatically for any binary package that contains executable binaries, and except for corner cases, it should not be necessary to use the old manually generated ones anymore. The package name for a automatic generated debug symbol package ends in -dbgsym.

The dbgsym packages are not installed into the regular archives, but in dedicated archives. That means, if you need the debug symbols for debugging, you need to add this archives to your apt configuration and then install the dbgsym package you are interested in. Please read https://wiki.debian.org/HowToGetABacktrace on how to do that.

6.9.9.2 Manual -dbg packages

Before the advent of the automatic dbgsym packages, debug packages needed to be manually generated. The name of a manual debug packages ends in -dbg. It is recommended to migrate such old legacy packages to the new dbgsym packages whenever possible. The procedure to convert your package is described in https://wiki.debian.org/AutomaticDebugPackages but the gist is to use the --dbgsym-migration='pkgname-dbg (<< currentversion~)' switch of the dh_strip command.

However, sometimes it is not possible to convert to the new dbgsym packages, or you will encounter the old manual -dbg packages in the archives, so you might need to deal with them. It is not recommended to create manual -dbg packages for new packages, except if the automatic ones won't work for some reason.

One reason could be that debug packages contains an entire special debugging build of a library or other binary. However, usually separating debugging information from the already built binaries is sufficient and will also save space and build time.

This is the case, for example, for debugging symbols of Python extensions. For now the right way to package Python extension debug symbols is to use -dbg packages as described in https://wiki.debian.org/Python/DbgBuilds.

To create -dbg packages, the package maintainer has to explicitly specify them in debian/control.

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

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

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

The dh_strip command in debhelper supports creating debug packages, and can take care of using objcopy to separate out the debugging symbols for you. If your package uses debhelper/9.20151219 or newer, you don't need to do anything. debhelper will generate debug symbol packages (as package-dbgsym) for you with no additional changes to your source package.

6.9.10 Best practices for meta-packages

A meta-package is a mostly empty package that makes it easy to install a coherent set of packages that can evolve over time. It achieves this by depending on all the packages of the set. Thanks to the power of APT, the meta-package maintainer can adjust the dependencies and the user's system will automatically get the supplementary packages. The dropped packages that were automatically installed will be also be marked as removal candidates (and are even automatically removed by aptitude). gnome and linux-image-amd64 are two examples of meta-packages (built by the source packages meta-gnome2 and linux-latest).

The long description of the meta-package must clearly document its purpose so that the user knows what they will lose if they remove the package. Being explicit about the consequences is recommended. This is particularly

important for meta-packages that are installed during initial installation and that have not been explicitly installed by the user. Those tend to be important to ensure smooth system upgrades and the user should be discouraged from uninstalling them to avoid potential breakages.

CHAPTER

SEVEN

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 in the Debian bug tracking system.

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 `https://bugs.debian.org/`*packagename*. Utilities like querybts 1 can also provide you with this information (and `reportbug` will usually invoke `querybts` before sending, too).

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

For extra credit, you can go through other packages, merging bugs which are reported more than once, or tagging bugs `fixed` when they have already been fixed. Note that when you are neither the bug submitter nor the package maintainer, you should not actually close the bug (unless you secure permission from the maintainer).

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

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

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

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

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

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

The program `mass-bug` (from the package `devscripts`) can be used to file bug reports against a list of packages.

7.1.1.1 Usertags

You may wish to use BTS usertags when submitting bugs across a number of packages. Usertags are similar to normal tags such as 'patch' and 'wishlist' but differ in that they are user-defined and occupy a namespace that is unique to a particular user. This allows multiple sets of developers to 'usertag' the same bug in different ways without conflicting.

To add usertags when filing bugs, specify the `User` and `Usertags` pseudo-headers:

```
To: submit@bugs.debian.org
Subject: title-of-bug

Package: pkgname
[ ... ]
User: email-addr
Usertags: tag-name [ tag-name ... ]

description-of-bug ...
```

Note that tags are separated by spaces and cannot contain underscores. If you are filing bugs for a particular group or team it is recommended that you set the `User` to an appropriate mailing list after describing your intention there.

To view bugs tagged with a specific usertag, visit `https://bugs.debian.org/cgi-bin/pkgreport.cgi?users=`*email-addr*`&tag=`*tag-name*.

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*) as possible. If you do not find that possible, then you should consider orphaning some of your packages (see *Orphaning a package*). Alternatively, you may ask the help of other people in order to catch up with the backlog of bugs that you have (you can ask for help on `debian-qa@lists.debian.org` or `debian-devel@lists.debian.org`). At the same time, you can look for co-maintainers (see *Collaborative maintenance*).

7.2.2 Bug squashing parties

From time to time the QA group organizes bug squashing parties to get rid of as many problems as possible. They are announced on `debian-devel-announce@lists.debian.org` and the announcement explains which area will be the focus of the party: usually they focus on release critical bugs but it may happen that they decide to help finish a major upgrade (like a new `perl` version that requires recompilation of all the binary modules).

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

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

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

7.3 Contacting other maintainers

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

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

You may also be interested in contacting the persons who are subscribed to a given source package via *The Debian Package Tracker*. You can do so by using the *package*`@packages.qa.debian.org` email address.

7.4 Dealing with inactive and/or unreachable maintainers

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

There is a simple system (the MIA database) in which information about maintainers who are deemed Missing In Action is recorded. When a member of the QA group contacts an inactive maintainer or finds more information about one, this is recorded in the MIA database. This system is available in `/org/qa.debian.org/mia` on the host `qa.debian.org`, and can be queried with the `mia-query` tool. Use `mia-query --help` to see how to query the database. If you find that no information has been recorded about an inactive maintainer yet, or that you can add more information, you should generally proceed as follows.

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

A non-functional e-mail address is a violation of Debian Policy. If an e-mail "bounces", please file a bug against the package and submit this information to the MIA database.

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

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

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

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

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

might even have died — you do not know who may be on the receiving side. Imagine how a relative will feel if they read the e-mail of the deceased and find a very impolite, angry and accusing message!

On the other hand, although we are volunteers, a package maintainer has made a commitment and therefore has a responsibility to maintain the package. 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 and/or interest.

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

7.5 Dealing with repeatedly negligent Debian developers

Debian developers are expected to comply with a series of technical and non-technical policies (for example, respectively, the Debian Machine Usage Policy and the Debian Code of Conduct).

To err is human and one should generally assume good intent, but repeated technical mistakes and/or disrespectful interactions, combined with a dismissive attitude towards Debian policies should be referred to the Debian Community Team. Most escalations to the Community Team are resolved amicably, but extreme cases may escalate further to the Debian Project Leader (leader@debian.org) and the Debian Account Managers (da-manager@debian.org>) and could ultimately lead to expulsion from the Debian project.

7.6 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.

7.6.1 Sponsoring packages

Sponsoring a package means uploading a package for a maintainer who is not able to do it on their own. It's not a trivial matter; the sponsor must verify the packaging and ensure that it is of the high level of quality that Debian strives to have.

Debian Developers can sponsor packages. Debian Maintainers can't.

The process of sponsoring a package is:

1. The maintainer prepares a source package (`.dsc`) and puts it online somewhere (like on mentors.debian.net) or even better, provides a link to a public VCS repository (see *salsa.debian.org: Git repositories and collaborative development platform*) where the package is maintained.
2. The sponsor downloads (or checks out) the source package.
3. The sponsor reviews the source package. If they find issues, they inform the maintainer and ask them to provide a fixed version (the process starts over at step 1).
4. The sponsor could not find any remaining problem. They build the package, sign it, and upload it to Debian.

Before delving into the details of how to sponsor a package, you should ask yourself whether adding the proposed package is beneficial to Debian.

There's no simple rule to answer this question; it can depend on many factors: is the upstream codebase mature and not full of security holes? Are there pre-existing packages that can do the same task and how do they compare to this new package? Has the new package been requested by users and how large is the user base? How active are the upstream developers?

You should also ensure that the prospective maintainer is going to be a good maintainer. Do they already have some experience with other packages? If yes, are they doing a good job with them (check out some bugs)? Are they familiar with the package and its programming language? Do they have the skills needed for this package? If not, are they able to learn them?

It's also a good idea to know where they stand with respect to Debian: do they agree with Debian's philosophy and do they intend to join Debian? Given how easy it is to become a Debian Member, you might want to only sponsor people who plan to join. That way you know from the start that you won't have to act as a sponsor indefinitely.

7.6.1.1 Sponsoring a new package

New maintainers usually have certain difficulties creating Debian packages — this is quite understandable. They will make mistakes. That's why sponsoring a brand new package into Debian requires a thorough review of the Debian packaging. Sometimes several iterations will be needed until the package is good enough to be uploaded to Debian. Thus being a sponsor implies being a mentor.

Don't ever sponsor a new package without reviewing it. The review of new packages done by ftpmasters mainly ensures that the software is really free. Of course, it happens that they stumble on packaging problems but they really should not. It's your task to ensure that the uploaded package complies with the Debian Free Software Guidelines and is of good quality.

Building the package and testing the software is part of the review, but it's also not enough. The rest of this section contains a non-exhaustive list of points to check in your review.[1]

- Verify that the upstream tarball provided is the same that has been distributed by the upstream author (when the sources are repackaged for Debian, generate the modified tarball yourself).
- Run `lintian` (see *lintian*). It will catch many common problems. Be sure to verify that any `lintian` overrides set up by the maintainer are fully justified.
- Run `licensecheck` (part of *devscripts*) and verify that `debian/copyright` seems correct and complete. Look for license problems (like files with "All rights reserved" headers, or with a non-DFSG compliant license). `grep -ri` is your friend for this task.
- Build the package with `pbuilder` (or any similar tool, see *pbuilder*) to ensure that the build-dependencies are complete.
- Proofread `debian/control`: does it follow the best practices (see *Best practices for debian/control*)? Are the dependencies complete?
- Proofread `debian/rules`: does it follow the best practices (see *Best practices for debian/rules*)? Do you see some possible improvements?
- Proofread the maintainer scripts (`preinst`, `postinst`, `prerm`, `postrm`, `config`): will the `preinst`/`postrm` work when the dependencies are not installed? Are all the scripts idempotent (i.e. can you run them multiple times without consequences)?
- Review any change to upstream files (either in `.diff.gz`, or in `debian/patches/` or directly embedded in the `debian` tarball for binary files). Are they justified? Are they properly documented (with DEP-3 for patches)?
- For every file, ask yourself why the file is there and whether it's the right way to achieve the desired result. Is the maintainer following the best packaging practices (see *Best Packaging Practices*)?
- Build the packages, install them and try the software. Ensure that you can remove and purge the packages. Maybe test them with `piuparts`.

If the audit did not reveal any problems, you can build the package and upload it to Debian. Remember that even if you're not the maintainer, as a sponsor you are still responsible for what you upload to Debian. That's why you're encouraged to keep up with the package through *The Debian Package Tracker*.

Note that you should not need to modify the source package to put your name in the `changelog` or in the `control` file. The `Maintainer` field of the `control` file and the `changelog` should list the person who did the packaging, i.e. the sponsee. That way they will get all the BTS mail.

Instead, you should instruct `dpkg-buildpackage` to use your key for the signature. You do that with the `-k` option:

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

If you use `debuild` and `debsign`, you can even configure it permanently in `~/.devscripts`:

[1] You can find more checks in the wiki, where several developers share their own sponsorship checklists.

```
DEBSIGN_KEYID=KEY-ID
```

7.6.1.2 Sponsoring an update of an existing package

You will usually assume that the package has already gone through a full review. So instead of doing it again, you will carefully analyze the difference between the current version and the new version prepared by the maintainer. If you have not done the initial review yourself, you might still want to have a deeper look just in case the initial reviewer was sloppy.

To be able to analyze the difference, you need both versions. Download the current version of the source package (with `apt-get source`) and rebuild it (or download the current binary packages with `aptitude download`). Download the source package to sponsor (usually with `dget`).

Read the new changelog entry; it should tell you what to expect during the review. The main tool you will use is `debdiff` (provided by the `devscripts` package); you can run it with two source packages (`.dsc` files), or two binary packages, or two `.changes` files (it will then compare all the binary packages listed in the `.changes`).

If you compare the source packages (excluding upstream files in the case of a new upstream version, for example by filtering the output of `debdiff` with `filterdiff -i '*/debian/*'`), you must understand all the changes you see and they should be properly documented in the Debian changelog.

If everything is fine, build the package and compare the binary packages to verify that the changes on the source package have no unexpected consequences (some files dropped by mistake, missing dependencies, etc.).

You might want to check out the Package Tracking System (see *The Debian Package Tracker*) to verify if the maintainer has not missed something important. Maybe there are translation updates sitting in the BTS that could have been integrated. Maybe the package has been NMUed and the maintainer forgot to integrate the changes from the NMU into their package. Maybe there's a release critical bug that they have left unhandled and that's blocking migration to `testing`. If you find something that they could have done (better), it's time to tell them so that they can improve for next time, and so that they have a better understanding of their responsibilities.

If you have found no major problem, upload the new version. Otherwise ask the maintainer to provide you a fixed version.

7.6.2 Granting upload permissions to DMs

After a Debian Maintainer's key has been added to the debian-maintainers keyring, a Debian Developer may grant upload permissions to the DM for specific packages by uploading a signed dak command to ftp.upload.debian.org as described in the FTP-Master's announcement to debian-devel.

This process can be simplified with the help of the `dcut` command from the `dput-ng` package. Note that this does not work with the `dcut` command from the `dput` package!

For example:

```
dcut dm --uid 0xfedcba9876543210 --allow nano --deny bash
```

If the DM's key is not in the keyring package yet but in the DD's local keyring, use the `--force` option and the fingerprint, without spaces and, in this special case, without the 0x prefix and in all uppercase:

```
dcut --force dm --uid FEDCBA9876543210FEDCBA9876543210 --allow nano
```

7.6.3 Advocating new developers

See the page about advocating a prospective developer at the Debian web site.

7.6.4 Handling new maintainer applications

Please see Checklist for Application Managers at the Debian web site.

CHAPTER

EIGHT

INTERNATIONALIZATION AND TRANSLATIONS

Debian supports an ever-increasing number of natural languages. Even if you are a native English speaker and do not speak any other language, it is part of your duty as a maintainer to be aware of issues of internationalization (abbreviated i18n because there are 18 letters between the 'i' and the 'n' in internationalization). Therefore, even if you are ok with English-only programs, you should read most of this chapter.

According to Introduction to i18n from Tomohiro KUBOTA, I18N (internationalization) means modification of software or related technologies so that software can potentially handle multiple languages, customs, and other differences, while L10N (localization) means implementation of a specific language for already-internationalized software.

l10n and i18n are interconnected, but the difficulties related to each of them are very different. It's not really difficult to allow a program to change the language in which texts are displayed based on user settings, but it is very time consuming to actually translate these messages. On the other hand, setting the character encoding is trivial, but adapting the code to use several character encodings is a really hard problem.

Setting aside the i18n problems, where no general guideline can be given, there is actually no central infrastructure for l10n within Debian which could be compared to the buildd mechanism for porting. So most of the work has to be done manually.

8.1 How translations are handled within Debian

Handling translation of the texts contained in a package is still a manual task, and the process depends on the kind of text you want to see translated.

For program messages, the gettext infrastructure is used most of the time. Often the translation is handled upstream within projects like the Free Translation Project, the GNOME Translation Project or the KDE Localization project. The only centralized resources within Debian are the Central Debian translation statistics, where you can find some statistics about the translation files found in the actual packages and download those files.

Package descriptions have translations since many years and Maintainers don't need to do anything special to support translated package descriptions; translators should use the Debian Description Translation Project (DDTP).

For `debconf` templates, maintainers should use the `po-debconf` package to ease the work of translators. Some statistics can be found on the Central Debian translation statistics site.

For web pages, each l10n team has access to the relevant VCS, and the statistics are available from the Central Debian translation statistics site.

For general documentation about Debian, the process is more or less the same as for the web pages (the translators have access to the VCS), but there are no statistics pages.

Another part of i18n work is package-specific documentation (man pages, info documents, other formats). At least the man page translations are po-based as most other things mentioned above.

8.2 I18N & L10N FAQ for maintainers

This is a list of problems that maintainers may face concerning i18n and l10n. While reading this, keep in mind that there is no real consensus on these points within Debian, and that this is only advice. If you have a better idea for a given problem, or if you disagree on some points, feel free to provide your feedback, so that this document can be enhanced.

8.2.1 How to get a given text translated

To translate package descriptions, you have nothing to do; the DDTP infrastructure will dispatch the material to translate to volunteers with no need for interaction on your part.

For all other material (`debconf` templates, gettext files, man pages, or other documentation), the best solution is to ask on debian-i18n for a translation in different languages. Some translation team members are subscribed to this list, and they will take care of the needed coordination, to get the material translated and reviewed. Once they are done, you will get your translated document from them in your mailbox or via a wishlist bugreport. It is also recommended, to use the `po-debconf` tools for i18n integration.

8.2.2 How to get a given translation reviewed

From time to time, individuals translate some texts in your package and will ask you for inclusion of the translation in the package. This can become problematic if you are not fluent in the given language. It is a good idea to send the document to the corresponding l10n mailing list, asking for a review. Once it has been done, you should feel more confident in the quality of the translation, and feel safe to include it in your package.

8.2.3 How to get a given translation updated

If you have some translations of a given text lying around, each time you update the original, you should ask the previous translator to update the translation with your new changes. Keep in mind that this task takes time; at least one week to get the update reviewed and all.

If the translator is unresponsive, you may ask for help on the corresponding l10n mailing list. If everything fails, don't forget to put a warning in the translated document, stating that the translation is somehow outdated, and that the reader should refer to the original document if possible.

Avoid removing a translation completely because it is outdated. Old documentation is often better than no documentation at all for non-English speakers.

8.2.4 How to handle a bug report concerning a translation

The best solution may be to mark the bug as forwarded to upstream, and forward it to both the previous translator and their team (using the corresponding debian-l10n-XXX mailing list).

8.3 I18N & L10N FAQ for translators

While reading this, please keep in mind that there is no general procedure within Debian concerning these points, and that in any case, you should collaborate with your team and the package maintainer.

8.3.1 How to help the translation effort

Choose what you want to translate, make sure that nobody is already working on it (using your debian-l10n-XXX mailing list), translate it, get it reviewed by other native speakers on your l10n mailing list, and provide it to the maintainer of the package (see next point).

8.3.2 How to provide a translation for inclusion in a package

Make sure your translation is correct (asking for review on your l10n mailing list) before providing it for inclusion. It will save time for everyone, and avoid the chaos resulting in having several versions of the same document in bug reports.

The best solution is to file a regular bug containing the translation against the package. Make sure to use both the `patch` and `l10n` tags, and to not use a severity higher than 'wishlist', since the lack of translation never prevented a program from running.

8.4 Best current practice concerning l10n

- As a maintainer, never edit the translations in any way (even to reformat the layout) without asking on the corresponding l10n mailing list. You risk for example breaking the encoding of the file by doing so. Moreover, what you consider an error can be right (or even needed) in the given language.
- As a translator, if you find an error in the original text, make sure to report it. Translators are often the most attentive readers of a given text, and if they don't report the errors they find, nobody will.
- In any case, remember that the major issue with l10n is that it requires several people to cooperate, and that it is very easy to start a flamewar about small problems because of misunderstandings. So if you have problems with your interlocutor, ask for help on the corresponding l10n mailing list, on debian-i18n, or even on debian-devel (but beware, l10n discussions very often become flamewars on that list :)
- In any case, cooperation can only be achieved with **mutual respect**.

CHAPTER

NINE

OVERVIEW OF DEBIAN MAINTAINER TOOLS

This section contains a rough overview of the tools available to maintainers. The following is by no means complete or definitive, but just a guide to some of the more popular tools.

Debian maintainer tools are meant to aid developers and free their time for critical tasks. As Larry Wall says, there's more than one way to do it.

Some people prefer to use high-level package maintenance tools and some do not. Debian is officially agnostic on this issue; any tool that gets the job done is fine. Therefore, this section is not meant to stipulate to anyone which tools they should use or how they should go about their duties of maintainership. Nor is it meant to endorse any particular tool to the exclusion of a competing tool.

Most of the descriptions of these packages come from the actual package descriptions themselves. Further information can be found in the package documentation itself. You can also see more info with the command `apt-cache show` *package-name*.

9.1 Core tools

The following tools are pretty much required for any maintainer.

9.1.1 dpkg-dev

`dpkg-dev` contains the tools (including `dpkg-source`) required to unpack, build, and upload Debian source packages. These utilities contain the fundamental, low-level functionality required to create and manipulate packages; as such, they are essential for any Debian maintainer.

9.1.2 debconf

`debconf` provides a consistent interface to configuring packages interactively. It is user interface independent, allowing end-users to configure packages with a text-only interface, an HTML interface, or a dialog interface. New interfaces can be added as modules.

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

Many feel that this system should be used for all packages that require interactive configuration; see *Configuration management with debconf*. `debconf` is not currently required by Debian Policy, but that may change in the future.

9.1.3 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, `dpkg-buildpackage` will use it automatically.

9.2 Package lint tools

According to the Free On-line Dictionary of Computing (FOLDOC), `lint` is: "A Unix C language processor which carries out more thorough checks on the code than is usual with C compilers." Package lint tools help package maintainers by automatically finding common problems and policy violations in their packages.

9.2.1 lintian

`lintian` dissects Debian packages and emits information about bugs and policy violations. It contains automated checks for many aspects of Debian policy as well as some checks for common errors.

You should periodically get the newest `lintian` from `unstable` and check over all your packages. Notice that the `-i` option provides detailed explanations of what each error or warning means, what its basis in Policy is, and commonly how you can fix the problem.

Refer to *Testing the package* 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 https://lintian.debian.org/. These reports contain the latest `lintian` output for the whole development distribution (`unstable`).

9.2.2 lintian-brush

`lintian-brush` contains a set of scripts that can automatically fix more than 80 common lintian issues in Debian packages.

It comes with a wrapper script that invokes the scripts, updates the changelog (if desired) and commits each change to version control.

9.2.3 piuparts

`piuparts` is the `.deb` package installation, upgrading, and removal testing tool.

`piuparts` tests that `.deb packages` handle installation, upgrading, and removal correctly. It does this by creating a minimal Debian installation in a chroot, and installing, upgrading, and removing packages in that environment, and comparing the state of the directory tree before and after. `piuparts` reports any files that have been added, removed, or modified during this process.

`piuparts` is meant as a quality assurance tool for people who create `.deb` packages to test them before they upload them to the Debian archive.

9.2.4 debdiff

`debdiff` (from the `devscripts` package, *devscripts*) compares file lists and control files of two packages. It is a simple regression test, as it will help you notice if the number of binary packages has changed since the last upload, or if something has changed in the control file. Of course, some of the changes it reports will be all right, but it can help you prevent various accidents.

You can run it over a pair of binary packages:

```
debdiff package_1-1_arch.deb package_2-1_arch.deb
```

Or even a pair of changes files:

```
debdiff package_1-1_arch.changes package_2-1_arch.changes
```

For more information please see debdiff 1.

9.2.5 diffoscope

`diffoscope` provides in-depth comparison of files, archives, and directories.

`diffoscope` will try to get to the bottom of what makes files or directories different. It will recursively unpack archives of many kinds and transform various binary formats into more human readable form to compare them.

Originally developed to compare two `.deb` files or two `changes` files nowadays it can compare two tarballs, ISO images, or PDF just as easily and supports a huge variety of filetypes.

The differences can be shown in a text or HTML report or as JSON output.

9.2.6 duck

duck, the Debian Url CheckKer, processes several fields in the debian/control, debian/upstream, debian/copyright, debian/patches/* and systemd.unit files and checks if URLs, VCS links and email address domains found therein are valid.

9.2.7 adequate

adequate checks packages installed on the system and reports policy violations.

The following checks are currently implemented:

- broken symlinks
- missing copyright file
- obsolete conffiles
- Python modules not byte-compiled
- missing libraries, undefined symbols, symbol size mismatches
- program name collisions
- missing alternatives
- missing binfmt interpreters and detectors
- missing pkg-config dependencies
- invalid user/group values in systemd/D-Bus/sysvinit files.

See /usr/share/doc/adequate/README on how to have adequate be ran automatically using autopkgtest.

9.2.8 i18nspector

i18nspector is a tool for checking translation templates (POT), message catalogues (PO) and compiled message catalogues (MO) files for common problems.

9.2.9 cme

cme is a tool from the libconfig-model-dpkg-perl package is an editor for dpkg source files with validation. Check the package description to see what it can do.

9.2.10 licensecheck

licensecheck attempts to determine the license that applies to each file passed to it, by searching the start of the file for text belonging to various licenses.

9.2.11 blhc

blhc is a tool which checks build logs for missing hardening flags.

9.3 Helpers for debian/rules

Package building tools make the process of writing debian/rules files easier. See *Helper scripts* for more information about why these might or might not be desired.

9.3.1 debhelper

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

Unlike some approaches, debhelper is broken into several small, simple commands, which act in a consistent manner. As such, it allows more fine-grained control than some of the other debian/rules tools.

There are a number of little debhelper add-on packages, too transient to document. You can see the list of most of them by doing apt-cache search ^dh-.

When choosing a debhelper compatibility level for your package, you should choose the highest compatibility level that is supported in the most recent stable release. Only use a higher compatibility level if you need specific features that are provided by that compatibility level that are not available in earlier levels.

In the past the compatibility level was defined in debian/compat, however nowadays it is much better to not use that but rather to use a versioned build-dependency like debhelper-compat (=12).

9.3.2 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.

9.3.3 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.

9.4 Package builders

The following packages help with the package building process, general driving of dpkg-buildpackage, as well as handling supporting tasks.

9.4.1 git-buildpackage

git-buildpackage provides the capability to inject or import Debian source packages into a Git repository, build a Debian package from the Git repository, and helps in integrating upstream changes into the repository.

These utilities provide an infrastructure to facilitate the use of Git by Debian maintainers. This allows one to keep separate Git branches of a package for stable, unstable and possibly experimental distributions, along with the other benefits of a version control system.

9.4.2 debootstrap

The debootstrap package and script allows you to bootstrap a Debian base system into any part of your filesystem. By base system, we mean the bare minimum of packages required to operate and install the rest of the system.

Having a system like this can be useful in many ways. For instance, you can chroot into it if you want to test your build dependencies. Or you can test how your package behaves when installed into a bare base system. Chroot builders use this package; see below.

9.4.3 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 cowbuilder, which speeds up the build process using a COW filesystem on any standard Linux filesystem.

9.4.4 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 *wanna-build* for more information, and https://buildd.debian.org/ to see the system in action.

9.5 Package uploaders

The following packages help automate or simplify the process of uploading packages into the official archive.

9.5.1 dupload

`dupload` is a package and a script to automatically upload Debian packages to the Debian archive, to log the upload, and to optionally send mail about the upload of a package. It supports various kinds of hooks to extend its functionality, and can be configured for new upload locations or methods, although by default it provides various hooks performing checks and comes configured with all Debian upload locations.

9.5.2 dput

The `dput` package and script do much the same thing as `dupload`, but in a different way. Out of the box it supports to run `dinstall` in dry-run mode after the upload.

9.5.3 dcut

The `dcut` script (part of the package `dput`, *dput*) helps in removing files from the ftp upload directory.

9.6 Maintenance automation

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

9.6.1 devscripts

`devscripts` is a package containing wrappers and tools that are very helpful for maintaining your Debian packages. Example scripts include `debchange` (or its alias, `dch`), which manipulates 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 (see https://wiki.debian.org/debian/watch for more info on that). `suspicious-source` outputs a list of files which are not common source files.

See the devscripts(7) manual page for a complete list of available scripts.

9.6.2 reportbug

`reportbug` is a tool designed to make the reporting of bugs in Debian and derived distributions relatively painless. Its features include:

- Integration with mutt and mh/nmh mail readers.
- Access to outstanding bug reports to make it easier to identify whether problems have already been reported.
- Automatic checking for newer versions of packages.

`reportbug` is designed to be used on systems with an installed mail transport agent; however, you can edit the configuration file and send reports using any available mail server.

This package also includes the `querybts` script for browsing the Debian bug tracking system.

9.6.3 autotools-dev

`autotools-dev` contains best practices for people who maintain 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.

9.6.4 dpkg-repack

`dpkg-repack` creates a 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 save the current state of a package before you upgrade it.

9.6.5 alien

`alien` converts binary packages between various packaging formats, including Debian, RPM (RedHat), LSB (Linux Standard Base), Solaris, and Slackware packages.

9.6.6 dpkg-dev-el

`dpkg-dev-el` is an Emacs lisp package that provides assistance when editing some of the files in the `debian` directory of your package. For instance, there are handy functions for listing a package's current bugs, and for finalizing the latest entry in a `debian/changelog` file.

9.6.7 dpkg-depcheck

`dpkg-depcheck` (from the `devscripts` package, *devscripts*) 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.

9.6.8 debputy

The `debputy` tools is new since 2024. While its main purpose is to offer a new Debian package build paradigm, it includes subcommands that can be used on any existing Debian package to validate the correctness of most of the files in `debian/*`, and in many cases also automatically fix them.

To check correctness of files in debian/* run:

```
debputy lint --spellcheck
```

To format debian/control and debian/tests/control files

```
debputy reformat --style black
```

Using the `reformat` command obsoletes using `wrap-and-sort -ast`.

The debputy tool also includes a language server which, when integrated with a code editor, can give real-time feedback on the correctness of files in `debian/*` while editing them.

For more information please see debputy 1.

9.7 Porting tools

The following tools are helpful for porters and for cross-compilation.

9.7.1 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.

9.8 Documentation and information

The following packages provide information for maintainers or help with building documentation.

9.8.1 debian-policy

The `debian-policy` package contains the Debian Policy Manual and related documents, which are:

- Debian Policy Manual
- Filesystem Hierarchy Standard (FHS)
- Debian Menu sub-policy
- Debian Perl sub-policy
- Debian configuration management specification
- Machine-readable debian/copyright specification
- Autopkgtest - automatic as-installed package testing
- Authoritative list of virtual package names
- Policy checklist for upgrading your packages

The Debian Policy Manual the policy relating to packages and details of the packaging mechanism. It covers everything from required `gcc` options to the way the maintainer scripts (`postinst` etc.) work, package sections and priorities, etc.

Also useful is the file `/usr/share/doc/debian-policy/upgrading-checklist.txt.gz`, which lists changes between versions of policy.

9.8.2 doc-debian

`doc-debian` contains lots of useful Debian-specific documentation:

- Debian Linux Manifesto
- Constitution for the Debian Project
- Debian Social Contract
- Debian Free Software Guidelines
- Debian Bug Tracking System documentation
- Introduction to the Debian mailing lists

9.8.3 developers-reference

The `developers-reference` package contains the document you are reading right now, the Debian Developer's Reference, a set of guidelines and best practices which has been established by and for the community of Debian developers.

9.8.4 maint-guide

The maint-guide package contains the Debian New Maintainers' Guide.

This document tries to describe the building of a Debian package to ordinary Debian users and prospective developers. It uses fairly non-technical language, and it's well covered with working examples.

9.8.5 debmake-doc

The debmake-doc package contains the Guide for Debian Maintainers.

This document is newer than Debian New Maintainers' Guide and intends to replace it. The Guide for Debian Maintainers caters to those learning Debian packaging and covers a wide range of topics and tools, along with plenty of examples about various types of packaging issues.

9.8.6 packaging-tutorial

This tutorial is an introduction to Debian packaging. It teaches prospective developers how to modify existing packages, how to create their own packages, and how to interact with the Debian community.

In addition to the main tutorial, it includes three practical sessions on modifying the grep package, and packaging the gnujump game and a Java library.

9.8.7 how-can-i-help

how-can-i-help shows opportunities for contributing to Debian. how-can-i-help hooks into APT to list opportunities for contributions to Debian (orphaned packages, bugs tagged 'newcomer') for packages installed locally, after each APT invocation. It can also be invoked directly, and then lists all opportunities for contribution (not just the new ones).

9.8.8 docbook-xml

docbook-xml provides the DocBook XML DTDs, which are commonly used for Debian documentation (as is the older debiandoc SGML DTD).

The docbook-xsl package provides the XSL files for building and styling the source to various output formats. You will need an XSLT processor, such as xsltproc, to use the XSL stylesheets. Documentation for the stylesheets can be found in the various docbook-xsl-doc-* packages.

To produce PDF from FO, you need an FO processor, such as xmlroff or fop. Another tool to generate PDF from DocBook XML is dblatex.

9.8.9 debiandoc-sgml

debiandoc-sgml provides the DebianDoc SGML DTD, which has been commonly used for Debian documentation, but is now deprecated (docbook-xml or python3-sphinx should be used instead).

9.8.10 debian-keyring

Contains the public OpenPGP keys of Debian Developers and Maintainers. See *Maintaining your public key* and the package documentation for more information.

9.8.11 debian-el

debian-el provides an Emacs mode for viewing Debian binary packages. This lets you examine a package without unpacking it.