debian

Guide for Debian Maintainers

Osamu Aoki

March 26, 2019

Guide for Debian Maintainers
by Osamu Aoki

Copyright © 2014-2017 Osamu Aoki

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the ”Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ”AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

This guide was made using the following previous documents as its reference:

- “Making a Debian Package (AKA the Debmake Manual)”, copyright © 1997 Jaldhar Vyas.
- “The New-Maintainer’s Debian Packaging Howto”, copyright © 1997 Will Lowe.
- “Debian New Maintainers’ Guide”, copyright © 1998-2002 Josip Rodin, 2005-2017 Osamu Aoki, 2010 Craig Small, and 2010 Raphaël Hertzog.

The latest version of this guide should be available:

- in the debmake-doc package and
- at the Debian Documentation web site.

Contents

Abstract

This “Guide for Debian Maintainers” (2019-03-26) tutorial guide describes the building of the Debian package to ordinary Debian users and prospective developers using the **debmake** command.

This guide focuses on the modern packaging style and comes with many simple examples.

- POSIX shell script packaging
- Python3 script packaging
- C with Makefile/Autotools/CMake
- multiple binary packages with shared library etc.

This “Guide for Debian Maintainers” can be considered as the successor to the “Debian New Maintainers’ Guide”.

Preface

If you are a somewhat experienced Debian user [1], you may have encountered following situations:

- You wish to install a certain software package not yet found in the Debian archive.
- You wish to update a Debian package with the newer upstream release.
- You wish to fix bugs of a Debian package with some patches.

If you wanted to create a Debian package to fulfill these wishes and to share your work with the community, you are the target audience of this guide as a prospective Debian maintainer. [2] Welcome to the Debian community.

Debian has many social and technical rules and conventions to follow since it is a large volunteer organization with history. Debian also has developed a huge array of packaging tools and archive maintenance tools to build consistent sets of binary packages addressing many technical objectives:

- packages build across many architectures (Section 5.4.4)
- reproducible build (Section 5.4.5)
- clean build under clearly specified package dependencies and patches (Section 5.5, Section 5.8, Section 7.10)
- optimal splits into multiple binary packages (Section 5.5.1)
- smooth library transitions (Section 5.18.2)
- interactive installation customization (Section 5.19)
- multiarch support (Section 5.20)
- security enhancement using specific compiler flags (Section 5.21)
- continuous integration (Section 5.22)
- boot strapping (Section 5.23)
- …

These make it somewhat overwhelming for many new prospective Debian maintainers to get involved with Debian. This guide tries to provide entry points for them to get started. It describes the following:

- What you should know before getting involved with Debian as a prospective maintainer.
- What it looks like to make a simple Debian package.
- What kind of rules exist for making the Debian package.
- Tips for making the Debian package.
- Examples of making Debian packages for several typical scenarios.

[1] You do need to know a little about Unix programming but you certainly don't need to be a wizard. You can learn about the basic handling of a Debian system from the Debian Reference. It contains some pointers to learn about Unix programming, too.

[2] If you are not interested in sharing the Debian package, you can certainly work around your local situation by compiling and installing the fixed upstream source package into **/usr/local/**.

The author felt limitations of updating the original “New Maintainers’ Guide” with the **dh-make** package and decided to create an alternative tool and its matching document to address modern requirements. The result is the **debmake** (version: 4.3.1) package and this updated “Guide for Debian Maintainers” in the **debmake-doc** (version: 1.14-1) package.

Many chores and tips have been integrated into the **debmake** command making this guide simple. This guide also offers many packaging examples.

Caution

It takes many hours to properly create and maintain Debian packages. The Debian maintainer must be **both technically competent and diligent** to take up this challenge.

Some important topics are explained in detail. Some of them may look irrelevant to you. Please be patient. Some corner cases are skipped. Some topics are only covered by the external pointers. These are intentional choices to keep this guide simple and maintainable.

Chapter 1

Overview

The Debian packaging of the *package-1.0*.**tar.gz**, containing a simple C source following the GNU Coding Standards and FHS, can be done with the **debmake** command as follows.

```
$ tar -xvzf package-1.0.tar.gz
$ cd package-1.0
$ debmake
  ... Make manual adjustments of generated configuration files
$ debuild
```

If manual adjustments of generated configuration files are skipped, the generated binary package lacks meaningful package description but still functions well under the **dpkg** command to be used for your local deployment.

> Caution
>
> 
>
> The **debmake** command only provides good template files. These template files must be manually adjusted to their perfection to comply with the strict quality requirements of the Debian archive, if the generated package is intended for general consumption.

If you are new to Debian packaging, do not worry about the details and just get the big picture instead.

If you have been exposed to Debian packaging, this looks very much like the **dh_make** command. This is because the **debmake** command is intended to replace functions offered historically by the **dh_make** command. [1]

The **debmake** command is designed with the following features:

- modern packaging style
  - **debian/copyright**: **DEP-5** compliant
  - **debian/control**: **substvar** support, **multiarch** support, multi binary packages, …
  - **debian/rules**: **dh** syntax, compiler hardening options, …
- flexibility
  - many options (Section 5.5.1.1, Chapter 6, Appendix A)
- sane default actions
  - execute non-stop with clean results
  - generate the multiarch package, unless the **-m** option is explicitly specified.
  - generate the non-native Debian package with the "**3.0 (quilt)**" format, unless the **-n** option is explicitly specified.

[1] The **deb-make** command was popular before the **dh_make** command. The current **debmake** package starts its version from **4.0** to avoid version overlaps with the obsolete **debmake** package, which provided the **deb-make** command.

- extra utility
  - verification of the **debian/copyright** file against the current source (Section 6.4)

The **debmake** command delegates most of the heavy lifting to its back-end packages: **debhelper**, **dpkg-dev**, **devscripts**, **pbuilder**, etc.

Tip

Make sure to protect the arguments of the **-b**, **-f**, **-I**, and **-w** options from shell interference by quoting them properly.

Tip

The non-native Debian package is the normal Debian package.

Tip

The detailed log of all the package build examples in this document can be obtained by following the instructions in Section 8.14.

Note

The generation of the **debian/copyright** file, and the outputs from the **-c** (Section 6.3) and **-k** (Section 6.4) options involve heuristic operations on the copyright and license information. They may produce some erroneous results.

Chapter 2

Prerequisites

Here are the prerequisites which you need to understand before you to get involved with Debian.

2.1 People around Debian

There are several types of people interacting around Debian with different roles:

- **upstream author**: the person who made the original program.
- **upstream maintainer**: the person who currently maintains the program.
- **maintainer**: the person making the Debian package of the program.
- **sponsor**: a person who helps maintainers to upload packages to the official Debian package archive (after checking their contents).
- **mentor**: a person who helps novice maintainers with packaging etc.
- **Debian Developer** (DD): a member of the Debian project with full upload rights to the official Debian package archive.
- **Debian Maintainer** (DM): a person with limited upload rights to the official Debian package archive.

Please note that you can't become an official **Debian Developer** (DD) overnight, because it takes more than technical skill. Please do not be discouraged by this. If it is useful to others, you can still upload your package either as a **maintainer** through a **sponsor** or as a **Debian Maintainer**.

Please note that you do not need to create any new packages to become an official Debian Developer. Contributing to the existing packages can provide a path to becoming an official Debian Developer too. There are many packages waiting for good maintainers (see Section 2.8).

2.2 How to contribute

Please refer to the following to learn how to contribute to Debian:

- How can you help Debian? (official)
- The Debian GNU/Linux FAQ, Chapter 13 - "Contributing to the Debian Project" (semi-official)
- Debian Wiki, HelpDebian (supplemental)
- Debian New Member site (official)
- Debian Mentors FAQ (supplemental)

2.3 Social dynamics of Debian

Please understand Debian's social dynamics to prepare yourself for interactions with Debian:

- We all are volunteers.
  - You can't impose on others what to do.
  - You should be motivated to do things by yourself.
- Friendly cooperation is the driving force.
  - Your contribution should not over-strain others.
  - Your contribution is valuable only when others appreciate it.
- Debian is not your school where you get automatic attention of teachers.
  - You should be able to learn many things by yourself.
  - Attention from other volunteers is a very scarce resource.
- Debian is constantly improving.
  - You are expected to make high quality packages.
  - You should adapt yourself to change.

Since we focus only on the technical aspects of the packaging in the rest of this guide, please refer to the following to understand the social dynamics of Debian:

- Debian: 17 years of Free Software, "do-ocracy", and democracy (Introductory slides by the ex-DPL)

2.4 Technical reminders

Here are some technical reminders to accommodate other maintainers to work on your package easily and effectively to maximize the output of Debian as a whole.

- Make your package easy to debug.
  - Keep your package simple.
  - Don't over-engineer your package.
- Keep your package well-documented.
  - Use readable code style.
  - Make comments in code.
  - Format code consistently.
  - Maintain the git repository [1] of the package.

Note

Debugging of software tends to consume more time than writing the initial working software.

[1] The overwhelming number of Debian maintainers use **git** over other VCS systems such as **hg**, **bzr**, etc.

2.5 Debian documentation

Please make yourself ready to read the pertinent part of the official Debian documentation together with this guide as needed to generate perfect Debian packages:

- "Debian Policy Manual"
  - "must follow" rules (https://www.debian.org/doc/devel-manuals#policy)
- "Debian Developer's Reference"
  - "best practice" document (https://www.debian.org/doc/devel-manuals#devref)

If this guide contradicts the official Debian documentation, the official Debian documentation is correct. Please file a bug report on the **debmake-doc** package using the **reportbug** command.

Here are alternative tutorial documents, which you may read along with this guide:

- "Debian New Maintainers' Guide" (older)
  - https://www.debian.org/doc/devel-manuals#maint-guide
  - https://packages.qa.debian.org/m/maint-guide.html
- "Debian Packaging Tutorial"
  - https://www.debian.org/doc/devel-manuals#packaging-tutorial
  - https://packages.qa.debian.org/p/packaging-tutorial.html
- "Ubuntu Packaging Guide" (Ubuntu is Debian based.)
  - http://packaging.ubuntu.com/html/

Tip

When reading these tutorials, you should consider using the **debmake** command in place of the **dh_make** command for better template files.

2.6 Help resources

Before you decide to ask your question in some public place, please do your part of the effort, i.e., read the fine documentation:

- package information available through the **aptitude**, **apt-cache**, and **dpkg** commands.
- files in **/usr/share/doc/***package* for all pertinent packages.
- contents of **man** *command* for all pertinent commands.
- contents of **info** *command* for all pertinent commands.
- contents of debian-mentors@lists.debian.org mailing list archive.
- contents of debian-devel@lists.debian.org mailing list archive.

Your desired information can be found effectively by using a well-formed search string such as "keyword **site:lists.debian.org**" to limit the search domain of the web search engine.

Making a small test package is a good way to learn details of the packaging. Inspecting existing well maintained packages is the best way to learn how other people make packages.

If you still have questions about the packaging, you can ask them interactively:

- debian-mentors@lists.debian.org mailing list. (This mailing list is for the novice.)
- debian-devel@lists.debian.org mailing list. (This mailing list is for the expert.)
- IRC such as #debian-mentors.
- Teams focusing on a specific set of packages. (Full list at https://wiki.debian.org/Teams)
- Language-specific mailing lists.
  - debian-devel-{french,italian,portuguese,spanish}@lists.debian.org
  - debian-chinese-gb@lists.debian.org (This mailing list is for general (Simplified) Chinese discussion.)
  - debian-devel@debian.or.jp

The more experienced Debian developers will gladly help you, if you ask properly after making your required efforts.

Caution

Debian development is a moving target. Some information found on the web may be outdated, incorrect, and non-applicable. Please use it carefully.

2.7 Archive situation

Please realize the situation of the Debian archive.

- Debian already has packages for most kinds of programs.
- The number of packages already in the Debian archive is several tens of times greater than that of active maintainers.
- Unfortunately, some packages lack an appropriate level of attention by the maintainer.

Thus, contributions to packages already in the archive are far more appreciated (and more likely to receive sponsorship for uploading) by other maintainers.

Tip

The **wnpp-alert** command from the **devscripts** package can check for installed packages up for adoption or orphaned.

2.8 Contribution approaches

Here is pseudo-Python code for your contribution approaches to Debian with a **program**:

```
if exist_in_debian(program):
  if is_team_maintained(program):
    join_team(program)
  if is_orphaned(program) # maintainer: Debian QA Group
    adopt_it(program)
  elif is_RFA(program) # Request for Adoption
    adopt_it(program)
  else:
    if need_help(program):
      contact_maintainer(program)
```

```
      triaging_bugs(program)
      preparing_QA_or_NMU_uploads(program)
    else:
      leave_it(program)
else: # new packages
  if not is_good_program(program):
    give_up_packaging(program)
  elif not is_distributable(program):
    give_up_packaging(program)
  else: # worth packaging
    if is_ITPed_by_others(program):
      if need_help(program):
        contact_ITPer_for_collaboration(program)
      else:
        leave_it_to_ITPer(program)
    else: # really new
      if is_applicable_team(program):
        join_team(program)
      if is_DFSG(program) and is_DFSG(dependency(program)):
        file_ITP(program, area="main") # This is Debian
      elif is_DFSG(program):
        file_ITP(program, area="contrib") # This is not Debian
      else: # non-DFSG
        file_ITP(program, area="non-free") # This is not Debian
      package_it_and_close_ITP(program)
```

Here:

- For exist_in_debian(), and is_team_maintained(); check:
  - the **aptitude** command
  - Debian packages web page
  - Teams
- For is_orphaned(), is_RFA(), and is_ITPed_by_others(); check:
  - The output of the **wnpp-alert** command.
  - Work-Needing and Prospective Packages
  - Debian Bug report logs: Bugs in pseudo-package wnpp in unstable
  - Debian Packages that Need Lovin'
  - Browse wnpp bugs based on debtags
- For is_good_program(), check:
  - The program should be useful.
  - The program should not introduce security and maintenance concerns to the Debian system.
  - The program should be well documented and its code needs to be understandable (i.e. not obfuscated).
  - The program's authors agree with the packaging and are amicable to Debian. [2]
- For is_it_DFSG(), and is_its_dependency_DFSG(); check:
  - Debian Free Software Guidelines (DFSG).
- For is_it_distributable(), check:
  - The software must have a license and it should allow its distribution.

You either need to file an **ITP** or adopt a package to start working on it. See the "Debian Developer's Reference":

- 5.1. New packages.
- 5.9. Moving, removing, renaming, orphaning, adopting, and reintroducing packages.

[2] This is not the absolute requirement. The hostile upstream may become a major resource drain for us all. The friendly upstream can be consulted to solve any problems with the program.

2.9 Novice contributor and maintainer

The novice contributor and maintainer may wonder what to learn to start your contribution to Debian. Here are my suggestions depending on your focus:

- Packaging
  - Basics of the **POSIX shell** and **make**.
  - Some rudimentary knowledge of **Perl** and **Python**.
- Translation
  - Basics of how the PO based translation system works.
- Documentation
  - Basics of text markups (XML, ReST, Wiki, …).

The novice contributor and maintainer may wonder where to start your contribution to Debian. Here are my suggestions depending on your skills:

- **POSIX shell**, **Perl**, and **Python** skills:
  - Send patches to the Debian Installer.
  - Send patches to the Debian packaging helper scripts such as **devscripts**, **pbuilder**, etc. mentioned in this document.
- **C** and **C++** skills:
  - Send patches to the packages with the **required** and **important** priorities.
- Non-English skills:
  - Send patches to the PO file of the Debian Installer.
  - Send patches to the PO file of the packages with the **required** and **important** priorities.
- Documentation skills:
  - Update contents on Debian Wiki.
  - Send patches to the existing Debian Documentation.

These activities should give you good exposure to the other Debian people to establish your credibility.
The novice maintainer should avoid packaging programs with the high security exposure:

- **setuid** or **setgid** program
- **daemon** program
- program installed in the **/sbin/** or **/usr/sbin/** directories

When you gain more experience in packaging, you'll be able to package such programs.

Chapter 3

Tool Setups

The **build-essential** package must be installed in the build environment.

The **devscripts** package should be installed in the maintainer environment.

Although this is not necessarily an absolute requirement, it is a good idea to install and set up all of the popular set of packages mentioned in this chapter in the maintainer environment. This enables us to share the common baseline working environment.

Please install the tools mentioned in the Overview of Debian Maintainer Tools in the "Debian Developer's Reference", as needed, too.

> Caution
>
> 
>
> Tool setups presented here are only meant as an example and may not be up-to-date with the latest packages on the system. Debian development is a moving target. Please make sure to read the pertinent documentation and update the configuration as needed.

3.1 Email address

Various Debian maintenance tools recognize your email address and name to use by the shell environment variables **$DEBEMAIL** and **$DEBFULLNAME**.

Let's setup these packages by adding the following lines to **~/.bashrc** [1].

Add to the ~/.bashrc file

```
DEBEMAIL="your.email.address@example.org"
DEBFULLNAME="Firstname Lastname"
export DEBEMAIL DEBFULLNAME
```

3.2 mc

The **mc** command offers very easy ways to manage files. It can open the binary **deb** file to check its content by pressing the Enter key over the binary **deb** file. It uses the **dpkg-deb** command as its back-end. Let's set it up to support easy **chdir** as follows.

Add to the ~/.bashrc file

```
# mc related
export HISTCONTROL=ignoreboth
. /usr/lib/mc/mc.sh
```

[1] This assumes you are using Bash as your login shell. If you use some other login shell such as Z shell, use their corresponding configuration files instead of **~/.bashrc**.

3.3 git

Nowadays, the **git** command is the essential tool to manage the source tree with history.

The global user configuration for the **git** command such as your name and email address can be set in **~/.gitconfig** as follows.

```
$ git config --global user.name "Name Surname"
$ git config --global user.email yourname@example.com
```

If you are too accustomed to the CVS or Subversion commands, you may wish to set several command aliases as follows.

```
$ git config --global alias.ci "commit -a"
$ git config --global alias.co checkout
```

You can check your global configuration as follows.

```
$ git config --global --list
```

Tip

It is essential to use some GUI git tools like **gitk** or **gitg** to work effectively with the history of the git repository.

3.4 quilt

The **quilt** command offers a basic method for recording modifications. For the Debian packaging, it should be customized to record modifications in the **debian/patches/** directory instead of its default **patches/** directory.

In order to avoid changing the behavior of the **quilt** command itself, let's create an alias **dquilt** for the Debian packaging by adding the following lines to the **~/.bashrc** file. The second line provides the same shell completion feature of the **quilt** command to the **dquilt** command.

Add to the ~/.bashrc file

```
alias dquilt="quilt --quiltrc=${HOME}/.quiltrc-dpkg"
complete -F _quilt_completion $_quilt_complete_opt dquilt
```

Then let's create **~/.quiltrc-dpkg** as follows.

```
d=.
while [ ! -d $d/debian -a `readlink -e $d` != / ];
    do d=$d/..; done
if [ -d $d/debian ] && [ -z $QUILT_PATCHES ]; then
    # if in Debian packaging tree with unset $QUILT_PATCHES
    QUILT_PATCHES="debian/patches"
    QUILT_PATCH_OPTS="--reject-format=unified"
    QUILT_DIFF_ARGS="-p ab --no-timestamps --no-index --color=auto"
    QUILT_REFRESH_ARGS="-p ab --no-timestamps --no-index"
    QUILT_COLORS="diff_hdr=1;32:diff_add=1;34:" + \
            "diff_rem=1;31:diff_hunk=1;33:diff_ctx=35:diff_cctx=33"
    if ! [ -d $d/debian/patches ]; then mkdir $d/debian/patches; fi
fi
```

See **quilt**(1) and How To Survive With Many Patches or Introduction to Quilt on how to use the **quilt** command.

See Section 4.8 for example usages.

3.5 devscripts

The **debsign** command, included in the **devscripts** package, is used to sign the Debian package with your private GPG key.

The **debuild** command, included in the **devscripts** package, builds the binary package and checks it with the **lintian** command. It is useful to have verbose outputs from the **lintian** command.

You can set these up in **~/.devscripts** as follows.

```
DEBUILD_DPKG_BUILDPACKAGE_OPTS="-i -I -us -uc"
DEBUILD_LINTIAN_OPTS="-i -I --show-overrides"
DEBSIGN_KEYID="Your_GPG_keyID"
```

The **-i** and **-I** options in **DEBUILD_DPKG_BUILDPACKAGE_OPTS** for the **dpkg-source** command help rebuilding of Debian packages without extraneous contents (see Section 5.15).

Currently, an RSA key with 4096 bits is a good idea. See Creating a new GPG key.

3.6 pbuilder

The **pbuilder** package provides a clean room (**chroot**) build environment. [2]

Let's customize it with several helper packages.

- The **cowbuilder** package to boost the chroot creation speed.
- The **lintian** package to find bugs in the package.
- The **bash**, **mc**, and **vim** packages in case build fails.
- The **ccache** package to boost the **gcc** speed. (optional)
- The **libeatmydata1** package to boost the **dpkg** speed. (optional)
- The parallel **make** to boost the build speed. (optional)

> Warning
>
> The optional customization may cause negative effects. In case of doubts, disable them.

Let's create **~/.pbuilderrc** as follows (all optional features are disabled).

```
AUTO_DEBSIGN="${AUTO_DEBSIGN:-no}"
PDEBUILD_PBUILDER=cowbuilder
HOOKDIR="/var/cache/pbuilder/hooks"
MIRRORSITE="http://deb.debian.org/debian/"
#APTCACHE=/var/cache/pbuilder/aptcache
APTCACHE=/var/cache/apt/archives
#BUILDRESULT=/var/cache/pbuilder/result/
BUILDRESULT=../
EXTRAPACKAGES="ccache lintian libeatmydata1"

# enable to use libeatmydata1 for pbuilder
#export LD_PRELOAD=${LD_PRELOAD+$LD_PRELOAD:}libeatmydata.so

# enable ccache for pbuilder
#export PATH="/usr/lib/ccache${PATH+:$PATH}"
#export CCACHE_DIR="/var/cache/pbuilder/ccache"
```

[2] The **sbuild** package provides an alternative chroot platform.

```
#BINDMOUNTS="${CCACHE_DIR}"

# parallel make
#DEBBUILDOPTS=-j8
```

Note

A symlink from **/root/.pbuilderrc** to **/home/<user>/.pbuilderrc** may help for the consistent experience.

Note

Due to Bug #606542, you may need to manually install packages listed in **EXTRAPACKAGES** into the chroot. See Section 7.10.

Note

Install **libeatmydata1** (>=82-2) both inside and outside of the chroot or disable to use **libeatmydata1**. This may cause a race condition with some build systems.

Note

The parallel **make** may fail for some existing packages and may make the build log difficult to read.

Let's create a hook scripts as follows.
/var/cache/pbuilder/hooks/A10ccache

```
#!/bin/sh
set -e
# increase the ccache caching size
ccache -M 4G
# output the current statistics
ccache -s
```

/var/cache/pbuilder/hooks/B90lintian

```
#!/bin/sh
set -e
apt-get -y --allow-downgrades install lintian
echo "+++ lintian output +++"
su -c "lintian -i -I --show-overrides /tmp/buildd/*.changes; :" -l pbuilder
echo "+++ end of lintian output +++"
```

/var/cache/pbuilder/hooks/C10shell

```
#!/bin/sh
set -e
apt-get -y --allow-downgrades install vim bash mc
# invoke shell if build fails
cd /tmp/buildd/*/debian/..
/bin/bash < /dev/tty > /dev/tty 2> /dev/tty
```

Note

All these scripts need to be set world executable: "**-rwxr-xr-x 1 root root**".

Note

The **ccache** cache directory **/var/cache/pbuilder/ccache** needs to be set world writable: "**-rwxrwxrwx 1 root root**" for the **pbuilder** command. You should be aware of associated security concerns.

3.7 git-buildpackage

You may wish to set several global configurations in **~/.gbp.conf**

```
# Configuration file for "gbp <command>"

[DEFAULT]
# the default build command:
builder = git-pbuilder -i -I -us -uc
# use pristine-tar:
pristine-tar = True
# Use color when on a terminal, alternatives: on/true, off/false or auto
color = auto
```

Tip

The **gbp** command is the alias of the **git-buildpackage** command.

3.8 HTTP proxy

You should set up a local HTTP caching proxy to save the bandwidth for the Debian package repository access. There are several choices:

- Simple HTTP caching proxy using the **squid** package.
- Specialized HTTP caching proxy using the **apt-cacher-ng** package.

3.9 Private Debian repository

You can set up a private Debian package repository with the **reprepro** package.

Chapter 4

Simple Example

There is an old Latin saying: "**Longum iter est per praecepta, breve et efficax per exempla**" ("It's a long way by the rules, but short and efficient with examples").

Here is an example of creating a simple Debian package from a simple C source using the **Makefile** as its build system.

Let's assume this upstream tarball to be **debhello-0.0.tar.gz**.

This type of source is meant to be installed as a non-system file as:

```
$ tar -xzmf debhello-0.0.tar.gz
$ cd debhello-0.0
$ make
$ make install
```

Debian packaging requires changing this "**make install**" process to install files to the target system image location instead of the normal location under **/usr/local**.

Note

Examples of creating a Debian package from other complicated build systems are described in Chapter 8.

4.1 Big picture

The big picture for building a single non-native Debian package from the upstream tarball **debhello-0.0.tar.gz** can be summarized as:

- The maintainer obtains the upstream tarball **debhello-0.0.tar.gz** and untars its contents to the **debhello-0.0** directory.
- The **debmake** command debianizes the upstream source tree by adding template files only in the **debian** directory.
  - The **debhello_0.0.orig.tar.gz** symlink is created pointing to the **debhello-0.0.tar.gz** file.
  - The maintainer customizes template files.
- The **debuild** command builds the binary package from the debianized source tree.
  - **debhello-0.0-1.debian.tar.xz** is created containing the **debian** directory.

Big picture of package building

```
$ tar -xzmf debhello-0.0.tar.gz
$ cd debhello-0.0
$ debmake
  ... manual customization
$ debuild
  ...
```

Tip

The **debuild** command in this and following examples may be substituted by equivalent commands such as the **pdebuild** command.

Tip

If the upstream tarball in the **.tar.xz** format is available, use it instead of the one in the **.tar.gz** and **.tar.bz2** formats. The **xz** compression format offers the better compression than the **gzip** and **bzip2** compressions.

4.2 What is debmake?

The **debmake** command is the helper script for the Debian packaging.

- It always sets most of the obvious option states and values to reasonable defaults.
- It generates the upstream tarball and its required symlink if they are missing.
- It doesn't overwrite the existing configuration files in the **debian/** directory.
- It supports the **multiarch** package.
- It creates good template files such as the **debian/copyright** file compliant with **DEP-5**.

These features make Debian packaging with **debmake** simple and modern.

Note

The **debmake** command isn't the only way to make a Debian package. Many packages are packaged using only a text editor while imitating how other similar packages are packaged.

4.3 What is debuild?

Here is a summary of commands similar to the **debuild** command.

- The **debian/rules** file defines how the Debian binary package is built.
- The **dpkg-buildpackage** command is the official command to build the Debian binary package. For normal binary build, it executes roughly:
  - "**dpkg-source --before-build**" (apply Debian patches, unless they are already applied)
  - "**fakeroot debian/rules clean**"

  - "**dpkg-source --build**" (build the Debian source package)
  - "**fakeroot debian/rules build**"
  - "**fakeroot debian/rules binary**"
  - "**dpkg-genbuildinfo**" (generate a ***.buildinfo** file)
  - "**dpkg-genchanges**" (generate a ***.changes** file)
  - "**fakeroot debian/rules clean**"
  - "**dpkg-source --after-build**" (unapply Debian patches, if they are applied during **--before-build**)
  - "**debsign**" (sign the ***.dsc** and ***.changes** files)
    * If you followed Section 3.5 to set the **-us** and **-uc** options, this step is skipped and you must run the **debsign** command manually.

- The **debuild** command is a wrapper script of the **dpkg-buildpackage** command to build the Debian binary package under the proper environment variables.
- The **pdebuild** command is a wrapper script to build the Debian binary package under the proper chroot environment with the proper environment variables.
- The **git-pbuilder** command is another wrapper script to build the Debian binary package under the proper chroot environment with the proper environment variables. This provides an easier command line UI to switch among different build environments.

Note

See **dpkg-buildpackage**(1) for exact details.

4.4 Step 1: Get the upstream source

Let's get the upstream source.

Download debhello-0.0.tar.gz

```
 $ wget http://www.example.org/download/debhello-0.0.tar.gz
 ...
 $ tar -xzmf debhello-0.0.tar.gz
 $ tree
.
├── debhello-0.0
│   ├── LICENSE
│   ├── Makefile
│   └── src
│       └── hello.c
└── debhello-0.0.tar.gz

2 directories, 4 files
```

Here, the C source **hello.c** is a very simple one.

hello.c

```
 $ cat debhello-0.0/src/hello.c
#include <stdio.h>
int
main()
{
        printf("Hello, world!\n");
```

```
        return 0;
}
```

Here, the **Makefile** supports GNU Coding Standards and FHS. Notably:

- build binaries honoring **$(CPPFLAGS)**, **$(CFLAGS)**, **$(LDFLAGS)**, etc.
- install files with **$(DESTDIR)** defined to the target system image
- install files with **$(prefix)** defined, which can be overridden to be **/usr**

Makefile

```
 $ cat debhello-0.0/Makefile
prefix = /usr/local

all: src/hello

src/hello: src/hello.c
        @echo "CFLAGS=$(CFLAGS)" | \
                fold -s -w 70 | \
                sed -e 's/^/# /'
        $(CC) $(CPPFLAGS) $(CFLAGS) $(LDCFLAGS) -o $@ $^

install: src/hello
        install -D src/hello \
                $(DESTDIR)$(prefix)/bin/hello

clean:
        -rm -f src/hello

distclean: clean

uninstall:
        -rm -f $(DESTDIR)$(prefix)/bin/hello

.PHONY: all install clean distclean uninstall
```

Note

The **echo** of the **$(CFLAGS)** variable is used to verify the proper setting of the build flag in the following example.

4.5 Step 2: Generate template files with debmake

Tip

If the **debmake** command is invoked with the **-T** option, more verbose comments are generated for the template files.

The output from the **debmake** command is very verbose and explains what it does as follows.

```
 $ cd debhello-0.0
 $ debmake
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="0.0", rev="1"
I: *** start packaging in "debhello-0.0". ***
I: provide debhello_0.0.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-0.0.tar.gz debhello_0.0.orig.tar.gz
I: pwd = "/path/to/debhello-0.0"
I: parse binary package settings:
I: binary package=debhello Type=bin / Arch=any M-A=foreign
I: analyze the source tree
I: build_type = make
I: scan source for copyright+license text and file extensions
I: 100 %, ext = c
I: check_all_licenses
I: ..
I: check_all_licenses completed for 2 files.
I: bunch_all_licenses
I: format_all_licenses
I: make debian/* template files
I: single binary package
I: debmake -x "1" ...
I: creating => debian/control
I: creating => debian/copyright
I: substituting => /usr/share/debmake/extra0/rules
I: creating => debian/rules
I: substituting => /usr/share/debmake/extra0/changelog
I: creating => debian/changelog
I: substituting => /usr/share/debmake/extra1/compat
I: creating => debian/compat
I: substituting => /usr/share/debmake/extra1/watch
I: creating => debian/watch
I: substituting => /usr/share/debmake/extra1/README.Debian
I: creating => debian/README.Debian
I: substituting => /usr/share/debmake/extra1source/local-options
I: creating => debian/source/local-options
I: substituting => /usr/share/debmake/extra1source/format
I: creating => debian/source/format
I: substituting => /usr/share/debmake/extra1patches/series
I: creating => debian/patches/series
I: run "debmake -x2" to get more template files
I: $ wrap-and-sort
```

The **debmake** command generates all these template files based on command line options. Since no options are specified, the **debmake** command chooses reasonable default values for you:

- The source package name: **debhello**
- The upstream version: **0.0**
- The binary package name: **debhello**
- The Debian revision: **1**
- The package type: **bin** (the ELF binary executable package)
- The **-x** option: **-x1** (default for the single binary package)

Let's inspect generated template files.

The source tree after the basic debmake execution.

```
 $ cd ..
 $ tree
.
├── debhello-0.0
│   ├── LICENSE
│   ├── Makefile
│   ├── debian
│   │   ├── README.Debian
│   │   ├── changelog
│   │   ├── compat
│   │   ├── control
│   │   ├── copyright
│   │   ├── patches
│   │   │   └── series
│   │   ├── rules
│   │   ├── source
│   │   │   ├── format
│   │   │   └── local-options
│   │   └── watch
│   └── src
│       └── hello.c
├── debhello-0.0.tar.gz
└── debhello_0.0.orig.tar.gz -> debhello-0.0.tar.gz

5 directories, 15 files
```

The **debian/rules** file is the build script provided by the package maintainer. Here is its template file generated by the **debmake** command.

debian/rules (template file):

```
 $ cat debhello-0.0/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@

#override_dh_auto_install:
#       dh_auto_install -- prefix=/usr

#override_dh_install:
#       dh_install --list-missing -X.pyc -X.pyo
```

This is essentially the standard **debian/rules** file with the **dh** command. (There are some commented out contents for you to customize it.)

The **debian/control** file provides the main meta data for the Debian package. Here is its template file generated by the **debmake** command.

debian/control (template file):

```
 $ cat debhello-0.0/debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: "Firstname Lastname" <email.address@example.org>
Build-Depends: debhelper (>=11~)
Standards-Version: 4.1.4
Homepage: <insert the upstream URL, if relevant>
```

```
Package: debhello
Architecture: any
Multi-Arch: foreign
Depends: ${misc:Depends}, ${shlibs:Depends}
Description: auto-generated package by debmake
 This Debian binary package was auto-generated by the
 debmake(1) command provided by the debmake package.
```

Warning

If you leave "**Section: unknown**" in the template **debian/control** file unchanged, the **lintian** error may cause the build to fail.

Since this is the ELF binary executable package, the **debmake** command sets "**Architecture: any**" and "**Multi-Arch: foreign**". Also, it sets required **substvar** parameters as "**Depends: ${shlibs:Depends}, ${misc:Depends}**". These are explained in Chapter 5.

Note

Please note this **debian/control** file uses the RFC-822 style as documented in 5.2 Source package control files — debian/control of the "Debian Policy Manual". The use of the empty line and the leading space are significant.

The **debian/copyright** file provides the copyright summary data of the Debian package. Here is its template file generated by the **debmake** command.

debian/copyright (template file):

```
 $ cat debhello-0.0/debian/copyright
Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: debhello
Source: <url://example.com>
#
# Please double check copyright with the licensecheck(1) command.

Files:     Makefile
           src/hello.c
Copyright: __NO_COPYRIGHT_NOR_LICENSE__
License:   __NO_COPYRIGHT_NOR_LICENSE__

#-------------------------------------------------------------------------...
# Files marked as NO_LICENSE_TEXT_FOUND may be covered by the following
# license/copyright files.

#-------------------------------------------------------------------------...
# License file: LICENSE
 License:
 .
 All files in this archive are licensed under the MIT License as below.
 .
 Copyright 2015 Osamu Aoki <osamu@debian.org>
 .
 Permission is hereby granted, free of charge, to any person obtaining a
 copy of this software and associated documentation files (the "Software"),
 to deal in the Software without restriction, including without limitation
 the rights to use, copy, modify, merge, publish, distribute, sublicense,
 and/or sell copies of the Software, and to permit persons to whom the
 Software is furnished to do so, subject to the following conditions:
```

```
 .
 The above copyright notice and this permission notice shall be included
 in all copies or substantial portions of the Software.
 .
 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
 OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
 IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
 CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

4.6 Step 3: Modification to the template files

Some manual modification is required to make the proper Debian package as a maintainer.

In order to install files as a part of the system files, the **$(prefix)** value of **/usr/local** in the **Makefile** should be overridden to be **/usr**. This can be accommodated by the following the **debian/rules** file with the **override_dh_auto_install** target setting "**prefix=/usr**".

debian/rules (maintainer version):

```
 $ vim debhello-0.0/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-0.0/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@

override_dh_auto_install:
        dh_auto_install -- prefix=/usr
```

Exporting the **DH_VERBOSE** environment variable in the **debian/rules** file as above forces the **debhelper** tool to make a fine grained build report.

Exporting **DEB_BUILD_MAINT_OPTION** as above sets the hardening options as described in the "FEATURE AREAS/ENVIRONMENT" in **dpkg-buildflags**(1). [1]

Exporting **DEB_CFLAGS_MAINT_APPEND** as above forces the C compiler to emit all the warnings.

Exporting **DEB_LDFLAGS_MAINT_APPEND** as above forces the linker to link only when the library is actually needed. [2]

The **dh_auto_install** command for the Makefile based build system essentially runs "**$(MAKE) install DESTDIR=debian/debhello**". The creation of this **override_dh_auto_install** target changes its behavior to "**$(MAKE) install DESTDIR=debian/debhello prefix=/usr**".

Here are the maintainer versions of the **debian/control** and **debian/copyright** files.

debian/control (maintainer version):

```
 $ vim debhello-0.0/debian/control
 ... hack, hack, hack, ...
 $ cat debhello-0.0/debian/control
Source: debhello
Section: devel
Priority: optional
```

[1] This is a cliché to force a read-only relocation link for the hardening and to prevent the lintian warning "**W: debhello: hardening-no-relro usr/bin/hello**". This is not really needed for this example but should be harmless. The lintian tool seems to produce a false positive warning for this case which has no linked library.

[2] This is a cliché to prevent overlinking for the complex library dependency case such as Gnome programs. This is not really needed for this simple example but should be harmless.

```
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>=11~)
Standards-Version: 4.3.0
Homepage: https://salsa.debian.org/debian/debmake-doc

Package: debhello
Architecture: any
Multi-Arch: foreign
Depends: ${misc:Depends}, ${shlibs:Depends}
Description: example package in the debmake-doc package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
```

debian/copyright (maintainer version):

```
 $ vim debhello-0.0/debian/copyright
 ... hack, hack, hack, ...
 $ cat debhello-0.0/debian/copyright
Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: debhello
Source: https://salsa.debian.org/debian/debmake-doc

Files:     *
Copyright: 2015 Osamu Aoki <osamu@debian.org>
License:   Expat
 Permission is hereby granted, free of charge, to any person obtaining a
 copy of this software and associated documentation files (the "Software"),
 to deal in the Software without restriction, including without limitation
 the rights to use, copy, modify, merge, publish, distribute, sublicense,
 and/or sell copies of the Software, and to permit persons to whom the
 Software is furnished to do so, subject to the following conditions:
 .
 The above copyright notice and this permission notice shall be included
 in all copies or substantial portions of the Software.
 .
 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
 OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
 IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
 CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

There are several other template files under the **debian/** directory. These also need to be updated.

Template files under debian/. (v=0.0):

```
 $ tree debhello-0.0/debian
debhello-0.0/debian
├── README.Debian
├── changelog
├── compat
├── control
├── copyright
├── patches
│   └── series
├── rules
├── source
│   ├── format
│   └── local-options
└── watch
```

```
2 directories, 10 files
```

Tip

Configuration files used by the **dh_*** commands from the **debhelper** package usually treat # as the start of a comment line.

4.7 Step 4: Building package with debuild

You can create a non-native Debian package using the **debuild** command or its equivalents (see Section 4.3) in this source tree. The command output is very verbose and explains what it does as follows.

```
 $ cd debhello-0.0
 $ debuild
 dpkg-buildpackage -us -uc -ui -i
 ...
 fakeroot debian/rules clean
dh clean
 ...
 debian/rules build
dh build
   dh_update_autotools_config
   dh_autoreconf
   dh_auto_configure
   dh_auto_build
        make -j4 "INSTALL=install --strip-program=true"
make[1]: Entering directory '/path/to/debhello-0.0'
# CFLAGS=-g -O2 -fdebug-prefix-map=/build/debmake-doc-1.14=.
# -fstack-protector-strong -Wformat -Werror=format-security
cc -Wdate-time -D_FORTIFY_SOURCE=2 -g -O2 -fdebug-prefix-map=/build/debmake-d...
 ...
 fakeroot debian/rules binary
dh binary
 ...
Now running lintian debhello_0.0-1_amd64.changes ...
 ...
W: debhello: binary-without-manpage usr/bin/hello
Finished running lintian.

 ...
```

You can verify that **CFLAGS** is updated properly with **-Wall** and **-pedantic** by the **DEB_CFLAGS_MAINT_APPEND** variable.

The manpage should be added to the package as reported by the **lintian** package, as shown in later examples (see Chapter 8). Let's move on for now.

Let's inspect the result.

The generated files of debhello version 0.0 by the debuild command:

```
 $ cd ..
 $ tree -FL 1
.
├── debhello-0.0/
├── debhello-0.0.tar.gz
├── debhello-dbgsym_0.0-1_amd64.deb
├── debhello_0.0-1.debian.tar.xz
```

```
├── debhello_0.0-1.dsc
├── debhello_0.0-1_amd64.build
├── debhello_0.0-1_amd64.buildinfo
├── debhello_0.0-1_amd64.changes
├── debhello_0.0-1_amd64.deb
└── debhello_0.0.orig.tar.gz -> debhello-0.0.tar.gz

1 directory, 9 files
```

You see all the generated files.

- The **debhello_0.0.orig.tar.gz** is a symlink to the upstream tarball.
- The **debhello_0.0-1.debian.tar.xz** contains the maintainer generated contents.
- The **debhello_0.0-1.dsc** is the meta data file for the Debian source package.
- The **debhello_0.0-1_amd64.deb** is the Debian binary package.
- The **debhello-dbgsym_0.0-1_amd64.deb** is the Debian debug symbol binary package. See Section 5.17.1.
- The **debhello_0.0-1_amd64.build** file is the build log file.
- The **debhello_0.0-1_amd64.buildinfo** file is the meta data file generated by **dpkg-genbuildinfo**(1).
- The **debhello_0.0-1_amd64.changes** is the meta data file for the Debian binary package.

The **debhello_0.0-1.debian.tar.xz** contains the Debian changes to the upstream source as follows.
The compressed archive contents of debhello_0.0-1.debian.tar.xz:

```
 $ tar -tzf debhello-0.0.tar.gz
debhello-0.0/
debhello-0.0/Makefile
debhello-0.0/src/
debhello-0.0/src/hello.c
debhello-0.0/LICENSE
 $ tar --xz -tf debhello_0.0-1.debian.tar.xz
debian/
debian/README.Debian
debian/changelog
debian/compat
debian/control
debian/copyright
debian/patches/
debian/patches/series
debian/rules
debian/source/
debian/source/format
debian/watch
```

The **debhello_0.0-1_amd64.deb** contains the binary files to be installed to the target system.
The **debhello-debsym_0.0-1_amd64.deb** contains the debug symbol files to be installed to the target system..
The binary package contents of all binary packages:

```
 $ dpkg -c debhello-dbgsym_0.0-1_amd64.deb
drwxr-xr-x root/root ...  ./
drwxr-xr-x root/root ...  ./usr/
drwxr-xr-x root/root ...  ./usr/lib/
drwxr-xr-x root/root ...  ./usr/lib/debug/
drwxr-xr-x root/root ...  ./usr/lib/debug/.build-id/
drwxr-xr-x root/root ...  ./usr/lib/debug/.build-id/8f/
-rw-r--r-- root/root ...  ./usr/lib/debug/.build-id/8f/6eac00576c538d13e7aea9...
drwxr-xr-x root/root ...  ./usr/share/
drwxr-xr-x root/root ...  ./usr/share/doc/
```

```
lrwxrwxrwx root/root ...  ./usr/share/doc/debhello-dbgsym -> debhello
 $ dpkg -c debhello_0.0-1_amd64.deb
drwxr-xr-x root/root ...  ./
drwxr-xr-x root/root ...  ./usr/
drwxr-xr-x root/root ...  ./usr/bin/
-rwxr-xr-x root/root ...  ./usr/bin/hello
drwxr-xr-x root/root ...  ./usr/share/
drwxr-xr-x root/root ...  ./usr/share/doc/
drwxr-xr-x root/root ...  ./usr/share/doc/debhello/
-rw-r--r-- root/root ...  ./usr/share/doc/debhello/README.Debian
-rw-r--r-- root/root ...  ./usr/share/doc/debhello/changelog.Debian.gz
-rw-r--r-- root/root ...  ./usr/share/doc/debhello/copyright
```

The generated dependency list of all binary packages.

The generated dependency list of all binary packages (v=0.0):

```
 $ dpkg -f debhello-dbgsym_0.0-1_amd64.deb pre-depends \
           depends recommends conflicts breaks
Depends: debhello (= 0.0-1)
 $ dpkg -f debhello_0.0-1_amd64.deb pre-depends \
           depends recommends conflicts breaks
Depends: libc6 (>= 2.2.5)
```

Caution

Many more details need to be addressed before uploading the package to the Debian archive.

Note

If manual adjustments of auto-generated configuration files by the **debmake** command are skipped, the generated binary package may lack meaningful package description and some of the policy requirements may be missed. This sloppy package functions well under the **dpkg** command, and may be good enough for your local deployment.

4.8 Step 3 (alternative): Modification to the upstream source

The above example did not touch the upstream source to make the proper Debian package.

An alternative approach as the maintainer is to change the upstream source by modifying the upstream **Makefile** to set the $(prefix) value to **/usr**.

The packaging is practically the same as the above step-by-step example except for two points in Section 4.6:

- Store the maintainer modifications to the upstream source as the corresponding patch files in the **debian/-patches/** directory and list their filenames in the **debian/patches/series** file as indicated in Section 5.8. There are several ways to generate patch files. A few examples are given in these sections:
  - Section 4.8.1
  - Section 4.8.2
  - Section 4.8.3
- The maintainer modification to the **debian/rules** file doesn't have the **override_dh_auto_install** target as follows:

debian/rules (alternative maintainer version):

```
 $ cd debhello-0.0
 $ vim debian/rules
 ... hack, hack, hack, ...
 $ cat debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@
```

This alternative approach to Debian packaging using a series of patch files may be less robust for future upstream changes but more flexible coping with the difficult upstream source. (See Section 7.13.)

Note

For this particular packaging case, the above Section 4.6 using the **debian/rules** file is the better approach. But let's keep on with this approach as a leaning process.

Tip

For more complicated packaging cases, both Section 4.6 and Section 4.8 approaches need to be deployed.

4.8.1 Patch by diff -u

Here is an example to create **000-prefix-usr.patch** by the **diff** command.

```
 $ cp -a debhello-0.0 debhello-0.0.orig
 $ vim debhello-0.0/Makefile
 ... hack, hack, hack, ...
 $ diff -Nru debhello-0.0.orig debhello-0.0 >000-prefix-usr.patch
 $ cat 000-prefix-usr.patch
diff -Nru debhello-0.0.orig/Makefile debhello-0.0/Makefile
--- debhello-0.0.orig/Makefile  2019-03-26 17:03:03.049996643 +0000
+++ debhello-0.0/Makefile       2019-03-26 17:03:03.121995074 +0000
@@ -1,4 +1,4 @@
-prefix = /usr/local
+prefix = /usr

 all: src/hello

 $ rm -rf debhello-0.0
 $ mv -f debhello-0.0.orig debhello-0.0
```

Please note that the upstream source tree is restored to the original state and the patch file is available as **000-prefix-usr.patch**.

This **000-prefix-usr.patch** is edited to be DEP-3 conformant and moved to the right location as below.

```
 $ cd debhello-0.0
 $ echo '000-prefix-usr.patch' >debian/patches/series
 $ vim ../000-prefix-usr.patch
 ... hack, hack, hack, ...
 $ mv -f ../000-prefix-usr.patch debian/patches/000-prefix-usr.patch
 $ cat debian/patches/000-prefix-usr.patch
From: Osamu Aoki <osamu@debian.org>
Description: set prefix=/usr patch
diff -Nru debhello-0.0.orig/Makefile debhello-0.0/Makefile
--- debhello-0.0.orig/Makefile
+++ debhello-0.0/Makefile
@@ -1,4 +1,4 @@
-prefix = /usr/local
+prefix = /usr

 all: src/hello
```

4.8.2 Patch by dquilt

Here is an example to create **000-prefix-usr.patch** by the **dquilt** command which is a simple wrapper of the **quilt** program. The syntax and function of the **dquilt** command is the same as the **quilt**(1) command, except for the fact that the patch is stored in the **debian/patches/** directory.

```
 $ cd debhello-0.0
 $ dquilt new 000-prefix-usr.patch
Patch debian/patches/000-prefix-usr.patch is now on top
 $ dquilt add Makefile
File Makefile added to patch debian/patches/000-prefix-usr.patch
 ... hack, hack, hack, ...
 $ head -1 Makefile
prefix = /usr
 $ dquilt refresh
Refreshed patch debian/patches/000-prefix-usr.patch
 $ dquilt header -e --dep3
 ... edit the DEP-3 patch header with editor
 $ tree -a
.
├── .pc
│   ├── .quilt_patches
│   ├── .quilt_series
│   ├── .version
│   ├── 000-prefix-usr.patch
│   │   ├── .timestamp
│   │   └── Makefile
│   └── applied-patches
├── LICENSE
├── Makefile
├── debian
│   ├── README.Debian
│   ├── changelog
│   ├── compat
│   ├── control
│   ├── copyright
│   ├── patches
│   │   ├── 000-prefix-usr.patch
│   │   └── series
│   ├── rules
│   ├── source
│   │   ├── format
│   │   └── local-options
│   └── watch
```

```
└── src
    └── hello.c

6 directories, 20 files
 $ cat debian/patches/series
000-prefix-usr.patch
 $ cat debian/patches/000-prefix-usr.patch
Description: set prefix=/usr patch
Author: Osamu Aoki <osamu@debian.org>
Index: debhello-0.0/Makefile
===================================================================
--- debhello-0.0.orig/Makefile
+++ debhello-0.0/Makefile
@@ -1,4 +1,4 @@
-prefix = /usr/local
+prefix = /usr

 all: src/hello
```

Here, **Makefile** in the upstream source tree doesn't need to be restored to the original state. The **dpkg-source** command invoked by the Debian packaging procedure in Section 4.7, understands the patch application state recorded by the **dquilt** program in the **.pc/** directory. As long as all the changes are committed by the **dquilt** command, the Debian source package can be built from the modified source tree.

Note

If the **.pc/** directory is missing, the **dpkg-source** command assumes that no patch was applied. That's why the more primitive patch generation methods like in Section 4.8.1 without generating the **.pc/** directory require the upstream source tree to be restored.

4.8.3 Patch by dpkg-source --commit

Here is an example to create **000-prefix-usr.patch** by the "**dpkg-source --commit**" command.

Let's edit the upstream source.

```
 $ cd debhello-0.0
 $ vim Makefile
 ... hack, hack, hack, ...
 $ head -n1 Makefile
prefix = /usr
```

Let's commit it.

```
 $ dpkg-source --commit . 000-prefix-usr.patch
... editor to edit the DEP-3 patch header
...
```

Let's see the result.

```
 $ cat debian/patches/series
000-prefix-usr.patch
 $ cat debian/patches/000-prefix-usr.patch
Description: set prefix=/usr patch
Author: Osamu Aoki <osamu@debian.org>
Index: debhello-0.0/Makefile

--- debhello-0.0.orig/Makefile
+++ debhello-0.0/Makefile
```

```
@@ -1,4 +1,4 @@
-prefix = /usr/local
+prefix = /usr

 all: src/hello

 $ tree -a
.
├── .pc
│   ├── .quilt_patches
│   ├── .quilt_series
│   ├── .version
│   ├── 000-prefix-usr.patch
│   │   ├── .timestamp
│   │   └── Makefile
│   └── applied-patches
├── LICENSE
├── Makefile
├── debian
│   ├── README.Debian
│   ├── changelog
│   ├── compat
│   ├── control
│   ├── copyright
│   ├── patches
│   │   ├── 000-prefix-usr.patch
│   │   └── series
│   ├── rules
│   ├── source
│   │   ├── format
│   │   └── local-options
│   └── watch
└── src
    └── hello.c

6 directories, 20 files
```

Here, the **dpkg-source** command performs exactly the same as what the sequences of the **dquilt** command did in Section 4.8.2.

Chapter 5

Basics

A broad overview is presented here for the basic rules of Debian packaging focusing on the non-native Debian package in the "**3.0 (quilt)**" format.

Note

Some details are intentionally skipped for clarity. Please read the manpages of the **dpkg-source**(1), **dpkg-buildpackage**(1), **dpkg**(1), **dpkg-deb**(1), **deb**(5), etc.

The Debian source package is a set of input files used to build the Debian binary package and is not a single file.

The Debian binary package is a special archive file which holds a set of installable binary data with its associated information.

A single Debian source package may generate multiple Debian binary packages defined in the **debian/control** file.

The non-native Debian package in the "**3.0 (quilt)**" format is the most normal Debian source package format.

Note

There are many wrapper scripts. Use them to streamline your workflow but make sure to understand the basics of their internals.

5.1 Packaging workflow

The Debian packaging workflow to create a Debian binary package involves generating several specifically named files (see Section 5.2) as defined in the "Debian Policy Manual".

The oversimplified method for the Debian packaging workflow can be summarized in 5 steps as follows.

1. The upstream tarball is downloaded as the *package-version***.tar.gz** file.
2. The upstream tarball is untarred to create many files under the *package-version*/ directory.
3. The upstream tarball is copied (or symlinked) to the particular filename *packagename_version***.orig.tar.gz**.
    - the character separating *package* and *version* is changed from - (hyphen) to **_** (underscore)
    - **.orig** is added in the file extension.
4. The Debian package specification files are added to the upstream source under the *package-version*/**debian/** directory.
    - Required specification files under the **debian/** directory:

debian/rules The executable script for building the Debian package (see Section 5.4)

debian/control The package configuration file containing the source package name, the source build dependencies, the binary package name, the binary dependencies, etc. (see Section 5.5)

debian/changelog The Debian package history file defining the upstream package version and the Debian revision in its first line (see Section 5.6)

debian/copyright The copyright and license summary (see Section 5.7)

- Optional specification files under the **debian/*** (see Section 5.11):
- The **debmake** command invoked in the *package-version/* directory provides the initial set of these template configuration files.
  - Required specification files are generated even with the **-x0** option.
  - The **debmake** command does not overwrite any existing configuration files.
- These files must be manually edited to their perfection according to the "Debian Policy Manual" and "Debian Developer's Reference".

5. The **dpkg-buildpackage** command (usually from its wrapper **debuild** or **pdebuild**) is invoked in the *package-version/* directory to make the Debian source and binary packages by invoking the **debian/rules** script.
   - The current directory is set as: **$(CURDIR)**=*/path/to/package-version/*
   - Create the Debian source package in the "**3.0 (quilt)**" format using **dpkg-source**(1)
     - *package_version*.**orig.tar.gz** (copy or symlink of *package-version*.**tar.gz**)
     - *package_version-revision*.**debian.tar.xz** (tarball of *package-version*/**debian/***)
     - *package_version-revision*.**dsc**
   - Build the source using "**debian/rules build**" into **$(DESTDIR)**
     - **DESTDIR=debian**/*binarypackage/* (single binary package)
     - **DESTDIR=debian/tmp/** (multi binary package)
   - Create the Debian binary package using **dpkg-deb**(1), **dpkg-genbuildinfo**(1), and **dpkg-genchanges**(1).
     - *binarypackage_version-revision_arch*.**deb**
     - … (There may be multiple Debian binary package files.)
     - *package_version-revision_arch*.**changes**
6. Check the quality of the Debian package with the **lintian** command. (recommended)
   - Follow the rejection guidelines from ftp-master.
     - REJECT-FAQ
     - NEW checklist
     - Lintian Autorejects (lintian tag list)
7. Sign the *package_version-revision*.**dsc** and *package_version-revision_arch*.**changes** files with the **debsign** command using your private GPG key.
8. Upload the set of the Debian source and binary package files with the **dput** command to the Debian archive.

Here, please replace each part of the filename as:

- the *package* part with the Debian source package name
- the *binarypackage* part with the Debian binary package name
- the *version* part with the upstream version
- the *revision* part with the Debian revision
- the *arch* part with the package architecture

Tip

Many patch management and VCS usage strategies for the Debian packaging are practiced. You don't need to use all of them.

Tip

There is very extensive documentation in Chapter 6. Best Packaging Practices in the "Debian Developer's Reference". Please read it.

5.1.1 The debhelper package

Although a Debian package can be made by writing a **debian/rules** script without using the **debhelper** package, it is impractical to do so. There are too many modern "Policy" required features to be addressed, such as application of the proper file permissions, use of the proper architecture dependent library installation path, insertion of the installation hook scripts, generation of the debug symbol package, generation of package dependency information, generation of the package information files, application of the proper timestamp for reproducible build, etc.

Debhelper package provides a set of useful scripts in order to simplify Debian's packaging workflow and reduce the burden of package maintainers. When properly used, they will help packagers handle and implement "Policy" required features automatically.

The modern Debian packaging workflow can be organized into a simple modular workflow by:

- using the **dh** command to invoke many utility scripts automatically from the **debhelper** package, and
- configuring their behavior with declarative configuration files in the **debian/** directory.

You should almost always use **debhelper** as your package's build dependency. This document also assumes that you are using a fairly contemporary version of **debhelper** to handle packaging works in the following contents.

5.2 Package name and version

If the upstream source comes as **hello-0.9.12.tar.gz**, you can take **hello** as the upstream source package name and **0.9.12** as the upstream version.

debmake is meant to provide template files for the package maintainer to work on. Comment lines started by **#** contain the tutorial text. You must remove or edit such comment lines before uploading to the Debian archive.

The license extraction and assignment process involves a lot of heuristics; it may fail in some cases. It is highly recommended to use other tools such as **licensecheck** from the **devscripts** package in conjunction with **debmake**.

There are some limitations for what characters may be used as a part of the Debian package. The most notable limitation is the prohibition of uppercase letters in the package name. Here is a summary as a set of regular expressions:

- Upstream package name (**-p**): [-+.a-z0-9]{2,}
- Binary package name (**-b**): [-+.a-z0-9]{2,}
- Upstream version (**-u**): [0-9][-+.:~a-z0-9A-Z]*
- Debian revision (**-r**): [0-9][+.~a-z0-9A-Z]*

See the exact definition in Chapter 5 - Control files and their fields in the "Debian Policy Manual".

debmake assumes relatively simple packaging cases. So all programs related to the interpreter are assumed to be "**Architecture: all**". This is not always true.

You must adjust the package name and upstream version accordingly for the Debian packaging.

In order to manage the package name and version information effectively under popular tools such as the **aptitude** command, it is a good idea to keep the length of package name to be equal or less than 30 characters; and the total length of version and revision to be equal or less than 14 characters. [1]

In order to avoid name collisions, the user visible binary package name should not be chosen from any generic words.

If upstream does not use a normal versioning scheme such as **2.30.32** but uses some kind of date such as **11Apr29**, a random codename string, or a VCS hash value as part of the version, make sure to remove them from the upstream version. Such information can be recorded in the **debian/changelog** file. If you need to invent a version string, use the **YYYYMMDD** format such as **20110429** as upstream version. This ensures that the **dpkg** command interprets later versions correctly as upgrades. If you need to ensure a smooth transition to a normal version scheme such as **0.1** in the future, use the **0~YYMMDD** format such as **0~110429** as upstream version, instead.

Version strings can be compared using the **dpkg** command as follows.

```
$ dpkg --compare-versions ver1 op ver2
```

The version comparison rule can be summarized as:

- Strings are compared from the head to the tail.
- Letters are larger than digits.
- Numbers are compared as integers.
- Letters are compared in ASCII code order.

There are special rules for period (**.**), plus (+), and tilde (~) characters, as follows.

```
0.0 < 0.5 < 0.10 < 0.99 < 1 < 1.0~rc1 < 1.0 < 1.0+b1 < 1.0+nmu1 < 1.1 < 2.0
```

One tricky case occurs when the upstream releases **hello-0.9.12-ReleaseCandidate-99.tar.gz** as the pre-release of **hello-0.9.12.tar.gz**. You can ensure the Debian package upgrade to work properly by renaming the upstream source to **hello-0.9.12~rc99.tar.gz**.

5.3 Native Debian package

The non-native Debian package in the "**3.0 (quilt)**" format is the most normal Debian source package format. The **debian/source/format** file should have "**3.0 (quilt)**" in it as described in **dpkg-source**(1). The above workflow and the following packaging examples always use this format.

A native Debian package is the rare Debian binary package format. It may be used only when the package is useful and valuable only for Debian. Thus, its use is generally discouraged.

Caution

A native Debian package is often accidentally built when its upstream tarball is not accessible from the **dpkg-buildpackage** command with its correct name *package_version*.**orig.tar.gz** . This is a typical newbie mistake caused by making a symlink name with "**-**" instead of the correct one with "_".

A native Debian package has no separation between the **upstream code** and the **Debian changes** and consists only of the following:

- *package_version*.**tar.gz** (copy or symlink of *package-version*.**tar.gz** with **debian/*** files.)
- *package_version*.**dsc**

[1] For more than 90% of packages, the package name is equal or less than 24 characters; the upstream version is equal or less than 10 characters and the Debian revision is equal or less than 3 characters.

If you need to create a native Debian package, create it in the "**3.0 (native)**" format using **dpkg-source**(1).

Tip

Some people promote packaging even programs that have been written only for Debian in the non-native package format. The required tarball without **debian/*** files needs to be manually generated in advance before the standard workflow in Section 5.1. [a] They claim that the use of non-native package format eases communication with the downstream distributions.

[a] Use of the "**debmake -t …**" command can help this workflow. See Section 6.2.

Tip

There is no need to create the tarball in advance if the native package format is used. The native Debian package can be created by setting the **debian/source/format** file to "**3.0 (native)**", setting the **debian/changelog** file to have the version without the Debian revision (**1.0** instead of **1.0-1**), and invoking the "**dpkg-source -b .**" command within the source tree. The tarball containing the source is generated by this.

5.4 debian/rules

The **debian/rules** script is the executable script to build the Debian package.

- The **debian/rules** script re-targets the upstream build system (see Section 5.16) to install files in the **$(DESTDIR)** and creates the archive file of the generated files as the **deb** file.
  - The **deb** file is used for the binary distribution and installed to the system using the **dpkg** command.
- The **dh** command is normally used as the front-end to the build system inside the **debian/rules** script.
- **$(DESTDIR)** path depends on the build type.
  - **$(DESTDIR)=debian/***binarypackage* (single binary package)
  - **$(DESTDIR)=debian/tmp** (multiple binary package)

5.4.1 dh

The **dh** command from the **debhelper** package with help from its associated packages functions as the wrapper to the typical upstream build systems and offers us uniform access to them by supporting all the Debian policy stipulated targets of the **debian/rules** file.

- **dh clean** : clean files in the source tree.
- **dh build** : build the source tree
- **dh build-arch** : build the source tree for architecture dependent packages
- **dh build-indep** : build the source tree for architecture independent packages
- **dh install** : install the binary files to **$(DESTDIR)**
- **dh install-arch** : install the binary files to **$(DESTDIR)** for architecture dependent packages
- **dh install-indep** : install the binary files to **$(DESTDIR)** for architecture independent packages
- **dh binary** : generate the **deb** file

- **dh binary-arch** : generate the **deb** file for architecture dependent packages
- **dh binary-indep** : generate the **deb** file for architecture independent packages

Note

For **debhelper** "compat >= 9", the **dh** command exports compiler flags (**CFLAGS**, **CXXFLAGS**, **FFLAGS**, **CPPFLAGS** and **LDFLAGS**) with values as returned by **dpkg-buildflags** if they are not set previously. (The **dh** command calls **set_buildflags** defined in the **Debian::Debhelper::Dh_Lib** module.)

5.4.2 Simple debian/rules

Thanks to this abstraction of the **dh** command [2], the Debian policy compliant **debian/rules** file supporting all the required targets can be written as simple as [3]:

Simple debian/rules:

```
#!/usr/bin/make -f
#export DH_VERBOSE = 1

%:
        dh $@
```

Essentially, this **dh** command functions as the sequencer to call all required **dh_*** commands at the right moment.

Note

The **debmake** command sets the **debian/control** file with "**Build-Depends: debhelper (>=9)**" and the **debian/compat** file with "**9**".

Tip

Setting "**export DH_VERBOSE = 1**" outputs every command that modifies files on the build system. Also it enables verbose build logs for some build systems.

5.4.3 Customized debian/rules

Flexible customization of the **debian/rules** script is realized by adding appropriate **override_dh_*** targets and their rules.

Whenever some special operation is required for a certain **dh_***foo* command invoked by the **dh** command, any automatic execution of it can be overridden by adding the makefile target **override_dh_***foo* in the **debian/rules** file.

The build process may be customized via the upstream provided interface such as arguments to the standard source build system commands, such as:

- **configure**,
- **Makefile**,

[2] This simplicity is available since version 7 of the **debhelper** package. This guide assumes the use of **debhelper** version 9 or newer.
[3] The **debmake** command generates a bit more complicated **debian/rules** file. But this is the core part.

- **setup.py**, or
- **Build.PL**.

If this is the case, you should add the **override_dh_auto_build** target and executing the "**dh_auto_build** -- *arguments*" command. This ensures passing *arguments* to the such build system after the default parameters that **dh_auto_build** usually passes.

Tip

Please try not to execute the above build system commands directly if they are supported by the **dh_auto_build** command.

The **debmake** command creates the initial template file taking advantage of the above simple **debian/rules** file example while adding some extra customizations for package hardening, etc. You need to know how underlying build systems work under the hood (see Section 5.16) to address their irregularities using package customization.

- See Section 4.6 for basic customization of the template **debian/rules** file generated by the **debmake** command.
- See Section 5.20 for multiarch customization.
- See Section 5.21 for hardening customization.

5.4.4 Variables for debian/rules

Some variable definitions useful for customizing **debian/rules** can be found in files under **/usr/share/dpkg/**. Notably:

pkg-info.mk **DEB_SOURCE**, **DEB_VERSION**, **DEB_VERSION_EPOCH_UPSTREAM**, **DEB_VERSION_UPSTREAM**, and **DEB_DISTRIBUTION** variables. These are useful for backport support etc..

vendor.mk **DEB_VENDOR** and **DEB_PARENT_VENDOR** variables; and **dpkg_vendor_derives_from** macro. These are useful for vendor support (Debian, Ubuntu, …).

architecture.mk Set **DEB_HOST_*** and **DEB_BUILD_*** variables. An alternative method of retrieving those variables is to invoke **dpkg-architecture** directly and query the value of a single variable. With explicit invocation of **dpkg-architecture** to retrieve necessary variables, there is no need to include **architecture.mk** in **debian/rules**, which would import all architecture-related variables.

buildflags.mk Set **CFLAGS**, **CPPFLAGS**, **CXXFLAGS**, **OBJCFLAGS**, **OBJCXXFLAGS**, **GCJFLAGS**, **FFLAGS**, **FCFLAGS**, and **LDFLAGS** build flags.

If you wish to use some of these useful variables in **debian/rules**, copy relevant code to **debian/rules** or write a simpler alternative in it. Please keep **debian/rules** simple.

For example, you can add an extra option to **CONFIGURE_FLAGS** for **linux-any** target architectures by adding the followings to **debian/rules**:

```
DEB_HOST_ARCH_OS ?= $(shell dpkg-architecture -qDEB_HOST_ARCH_OS)
 ...
ifeq ($(DEB_HOST_ARCH_OS),linux)
CONFIGURE_FLAGS += --enable-wayland
endif
```

Tip

It was useful to include **buildflags.mk** in **debian/rules** to set the build flags such as **CPPFLAGS**, **CFLAGS**, **LDFLAGS**, etc. properly while honoring **DEB_CFLAGS_MAINT_APPEND**, **DEB_BUILD_MAINT_OPTIONS**, etc. for the **debhelper** "compat <= 8". Now you should use the **debhelper** "compat >= 9", should not include **buildflags.mk** without specific reasons, and should let the **dh** command set these build flags.

See Section 5.20, **dpkg-architecture**(1) and **dpkg-buildflags**(1).

5.4.5 Reproducible build

Here are some recommendations to attain a reproducible build result.

- Don't embed the timestamp based on the system time.
- Use "**dh $@**" in the **debian/rules** to access the latest **debhelper** features.
- Export the build environment as "**LC_ALL=C.UTF-8**" (see Section 7.15).
- Set the timestamp used in the upstream source from the value of the debhelper-provided environment variable **$SOURCE_DATE_EPOCH**.
- Read more at ReproducibleBuilds.
  - ReproducibleBuilds Howto.
  - ReproducibleBuilds TimestampsProposal.

The control file *source-name_source-version_arch*.**buildinfo** generated by **dpkg-genbuildinfo**(1) records the build environment. See **deb-buildinfo**(5)

5.5 debian/control

The **debian/control** file consists of blocks of meta data separated by a blank line. Each block of meta data defines the following in this order:

- meta data for the Debian source package
- meta data for the Debian binary packages

See Chapter 5 - Control files and their fields of the "Debian Policy Manual" for the definition of each meta data.

5.5.1 Split of a Debian binary package

For well behaving build systems, the split of a Debian binary package into small ones can be realized as follows.

- Create binary package entries for all binary packages in the **debian/control** file.
- List all file paths (relative to **debian/tmp**) in the corresponding **debian/***binarypackage***.install** files.

Please check examples in this guide:

- Section 8.11 (Autotools based)
- Section 8.12 (CMake based)

5.5.1.1 debmake -b

The **debmake** command with the **-b** option provides an intuitive and flexible method to create the initial template **debian/control** file defining the split of the Debian binary packages with following stanzas:

- **Package:**
- **Architecture:**
- **Multi-Arch:**
- **Depends:**
- **Pre-Depends:**

The **debmake** command also sets an appropriate set of substvars used in each pertinent dependency stanza. Let's quote the pertinent part from the **debmake** manpage here.

-b ***"binarypackage[:type],…",*** **--binaryspec** ***"binarypackage[:type],…"*** set the binary package specs by a comma separated list of *binarypackage:type* pairs, e.g., in the full form "**foo:bin,foo-doc:doc,libfoo1:lib,libfoo-dev:dev**" or in the short form, "**-doc,libfoo1,libfoo-dev**".

Here, *binarypackage* is the binary package name, and the optional *type* is chosen from the following *type* values:

- **bin**: C/C++ compiled ELF binary code package (any, foreign) (default, alias: ***""***, i.e., *null-string*)
- **data**: Data (fonts, graphics, …) package (all, foreign) (alias: **da**)
- **dev**: Library development package (any, same) (alias: **de**)
- **doc**: Documentation package (all, foreign) (alias: **do**)
- **lib**: Library package (any, same) (alias: **l**)
- **perl**: Perl script package (all, foreign) (alias: **pl**)
- **python**: Python script package (all, foreign) (alias: **py**)
- **python3**: Python3 script package (all, foreign) (alias: **py3**)
- **ruby**: Ruby script package (all, foreign) (alias: **rb**)
- **script**: Shell script package (all, foreign) (alias: **sh**)

The pair values in the parentheses, such as (any, foreign), are the **Architecture** and **Multi-Arch** stanza values set in the **debian/control** file.

In many cases, the **debmake** command makes good guesses for *type* from *binarypackage*. If *type* is not obvious, *type* is set to **bin**. For example, **libfoo** sets *type* to **lib**, and **font-bar** sets *type* to **data**, …

If the source tree contents do not match settings for *type*, the **debmake** command warns you.

5.5.1.2 Package split scenario and examples

Here are some typical multiarch package split scenarios for the following upstream source examples using the **debmake** command:

- a library source *libfoo-1.0***.tar.gz**
- a tool source *bar-1.0***.tar.gz** written in a compiled language
- a tool source *baz-1.0***.tar.gz** written in an interpreted language

binarypackage	***type***	**Architecture:**	**Multi-Arch:**	**Package content**
lib*foo1*	**lib***	any	same	the shared library, co-installable
lib*foo*-**dev**	**dev***	any	same	the shared library header files etc., co-installable
lib*foo*-**tools**	**bin***	any	foreign	the run-time support programs, not co-installable

binarypackage	***type***	**Architecture:**	**Multi-Arch:**	**Package content**
lib*foo*-**doc**	**doc***	all	foreign	the shared library documentation files
bar	**bin***	any	foreign	the compiled program files, not co-installable
bar-**doc**	**doc***	all	foreign	the documentation files for the program
baz	**script**	all	foreign	the interpreted program files

5.5.1.3 The library package name

Let's consider that the upstream source tarball of the **lib***foo* library is updated from **lib***foo-7.0***.tar.gz** to **lib***foo-8.0***.tar.gz** with a new SONAME major version which affects other packages.

The binary library package must be renamed from **lib***foo7* to **lib***foo8* to keep the **unstable** suite system working for all dependent packages after the upload of the package based on the **lib***foo-8.0***.tar.gz**.

Warning

If the binary library package isn't renamed, many dependent packages in the **unstable** suite become broken just after the library upload even if a binNMU upload is requested. The binNMU may not happen immediately after the upload due to several reasons.

The **-dev** package must follow one of the following naming rules:

- Use the **unversioned -dev** package name: **lib***foo*-**dev**
  - This is the typical one for leaf library packages.
  - Only one version of the library source package is allowed in the archive.
    - The associated library package needs to be renamed from **lib***foo7* to **lib***foo8* to prevent dependency breakage in the **unstable** archive during the library transition.
  - This approach should be used if the simple binNMU resolves the library dependency quickly for all affected packages. (ABI change by dropping the deprecated API while keeping the active API unchanged.)
  - This approach may still be a good idea if manual code updates, etc. can be coordinated and manageable within limited packages. (API change)
- Use the **versioned -dev** package names: **lib***foo7***-dev** and **lib***foo8***-dev**
  - This is typical for many major library packages.
  - Two versions of the library source packages are allowed simultaneously in the archive.
    - Make all dependent packages depend on **lib***foo*-**dev**.
    - Make both **lib***foo7***-dev** and **lib***foo8***-dev** provide **lib***foo*-**dev**.
    - The source package needs to be renamed as **lib***foo7-7.0***.tar.gz** and **lib***foo8-8.0***.tar.gz** respectively from **lib***foo-?.0***.tar.gz**.
    - The package specific install file path including **lib***foo7* and **lib***foo8* respectively for header files etc. needs to be chosen to make them co-installable.
  - Do not use this heavy handed approach, if possible.
  - This approach should be used if the update of multiple dependent packages require manual code updates, etc. (API change) Otherwise, the affected packages become RC buggy with FTBFS.

Tip

If the data encoding scheme changes (e.g., latin1 to utf-8), the same care as the API change needs to be taken.

See Section 5.18.

5.5.2 Substvar

The **debian/control** file also defines the package dependency in which the variable substitutions mechanism (substvar) may be used to free package maintainers from chores of tracking most of the simple package dependency cases. See **deb-substvars**(5).

The **debmake** command supports the following substvars:

- **${misc:Depends}** for all binary packages
- **${misc:Pre-Depends}** for all multiarch packages
- **${shlibs:Depends}** for all binary executable and library packages
- **${python:Depends}** for all Python packages
- **${python3:Depends}** for all Python3 packages
- **${perl:Depends}** for all Perl packages
- **${ruby:Depends}** for all Ruby packages

For the shared library, required libraries found simply by "**objdump -p** */path/to/program* **| grep NEEDED**" are covered by the **shlib** substvar.

For Python and other interpreters, required modules found simply looking for lines with "**import**", "**use**", "**require**", etc., are covered by the corresponding substvars.

For other programs which do not deploy their own substvars, the **misc** substvar covers their dependency.

For POSIX shell programs, there is no easy way to identify the dependency and no substvar covers their dependency.

For libraries and modules required via the dynamic loading mechanism including the GObject introspection mechanism, there is no easy way to identify the dependency and no substvar covers their dependency.

5.5.3 binNMU safe

A binNMU is a binary-only non-maintainer upload performed for library transitions etc. In a binNMU upload, only the "**Architecture: any**" packages are rebuilt with a suffixed version number (e.g. version 2.3.4-3 will become 2.3.4-3+b1). The "**Architecture: all**" packages are not built.

The dependency defined in the **debian/control** file among binary packages from the same source package should be safe for the binNMU. This needs attention if there are both "**Architecture: any**" and "**Architecture: all**" packages involved in it.

- "**Architecture: any**" package: depends on "**Architecture: any**" *foo* package
  - **Depends:** *foo* **(= ${binary:Version})**
- "**Architecture: any**" package: depends on "**Architecture: all**" *bar* package
  - **Depends:** *bar* **(= ${source:Version}**
- "**Architecture: all**" package: depends on "**Architecture: any**" *baz* package
  - **Depends:** *baz* **(>= ${source:Version})**, *baz* **(<< ${source:Upstream-Version}.0~)**

5.6 debian/changelog

The **debian/changelog** file records the Debian package history and defines the upstream package version and the Debian revision in its first line. The changes need to be documented in the specific, formal, and concise style.

- Even if you are uploading your package by yourself, you must document all non-trivial user-visible changes such as:
  - the security related bug fixes.

  - the user interface changes.

- If you are asking your sponsor to upload it, you should document changes more comprehensively, including all packaging related ones, to help reviewing your package.
  - The sponsor shouldn't second guess your thought behind your package.
  - The sponsor's time is more valuable than yours.

The **debmake** command creates the initial template file with the upstream package version and the Debian revision. The distribution is set to **UNRELEASED** to prevent accidental upload to the Debian archive.

The **debchange** command (alias **dch**) is commonly used to edit this.

Tip

You can edit the **debian/changelog** file manually with any text editor as long as you follow the formatting convention used by the **debchange** command.

Tip

The date string used in the **debian/changelog** file can be manually generated by the "**LC_ALL=C date -R**" command.

This is installed in the **/usr/share/doc/***binarypackage* directory as **changelog.Debian.gz** by the **dh_installchangelogs** command.

The upstream changelog is installed in the **/usr/share/doc/***binarypackage* directory as **changelog.gz**.

The upstream changelog is automatically found by the **dh_installchangelogs** using the case insensitive match of its file name to **changelog**, **changes**, **changelog.txt**, **changes.txt**, **history**, **history.txt**, or **changelog.md** and searched in the **./ doc/** or **docs/** directories.

After finishing your packaging and verifying its quality, please execute the "**dch -r**" command and save the finalized **debian/changelog** file with the distribution normally set to **unstable**. [4] If you are packaging for backports, security updates, LTS, etc., please use the appropriate distribution names instead.

5.7 debian/copyright

Debian takes the copyright and license matters very seriously. The "Debian Policy Manual" enforces having a summary of them in the **debian/copyright** file in the package.

You should format it as a machine-readable debian/copyright file (DEP-5).

Caution

The **debian/copyright** file should be sorted to keep the generic file patterns at the top of the list. See Section 6.4.

The **debmake** command creates the initial DEP-5 compatible template file by scanning the entire source tree. It uses an internal license checker to classify each license text. [5]

Unless specifically requested to be pedantic with the **-P** option, the **debmake** command skips reporting for auto-generated files with permissive licenses to be practical.

[4] If you are using the **vim** editor, make sure to save this with the "**:wq**" command.

[5] The **licensecheck** command from the **devscripts** package was referenced to make this internal checker. Now the **licensecheck** command is provided in an independent **licensecheck** package with a lot of improvements.

Note

If you find issues with this license checker, please file a bug report to the **debmake** package with the problematic part of text containing the copyright and license.

Note

The **debmake** command focuses on bunching up same copyright and license claims in detail to create template for **debian/copyright**. In order to do this within reasonable time, it only picks the first section which looks like copyright and license claims. So its license assignment may not be optimal. Please also use other tools such as **licensecheck**.

Tip

You are highly encouraged to check the license status with the **licensecheck**(1) command and, as needed, with your manual code review.

5.8 debian/patches/*

The **-p1** patches in the **debian/patches/** directory are applied in the sequence defined in the **debian/patches/series** file to the upstream source tree before the **build** process.

Note

The native Debian package (see Section 5.3) doesn't use these files.

There are several methods to prepare a series of **-p1** patches.

- The **diff** command
  - See Section 4.8.1
  - Primitive but versatile method
    - Patches may come from other distros, mailing list postings, or cherry-picked patches from the upstream **git** repository with the "**git format-patches**" command
  - Missing the **.pc/** directory
  - Unmodified upstream source tree
  - Manually update the **debian/patches/series** file
- The **dquilt** command
  - See Section 3.4
  - Basic convenient method
  - Proper generation of the **.pc/** directory data
  - Modified upstream source tree

- The "**dpkg-source --commit**" command
  - See Section 4.8.3
  - Newer elegant method
  - Proper generation of the **.pc/** directory data
  - Modified upstream source tree
- The automatic patch generation by the **dpkg-buildpackage**
  - See Section 5.14
  - Add **single-debian-patch** in the **debian/source/local-options** file
  - Set the **debian/source/local-patch-header** file
  - Missing the **.pc/** directory
  - Modified upstream source tree in the Debian branch (**master**)
- The **gbp-pq** command
  - basic **git** work flow with the **git-buildpackage** package
  - Missing the **.pc/** directory
  - Modified upstream source tree in the throw-away branch (**patch-queue/master**)
  - Unmodified upstream source tree in the Debian branch (**master**)
- The **gbp-dpm** command
  - more elaborate **git** work flow with the **git-dpm** package
  - Missing the **.pc/** directory
  - Modified upstream source tree in the patched branch (**patched**/*whatever*)
  - Unmodified upstream source tree in the Debian branch (**master**/*whatever*)

Wherever these patches come from, it is a good idea to tag them with a DEP-3 compatible header.

Tip

The **dgit** package offers an alternative git integration tool with the Debian package archive.

5.8.1 dpkg-source -x

The "**dpkg-source -x**" command unpacks the Debian source package.

It normally applies the patches in the **debian/patches/** directory to the source tree and records the patch state in the **.pc/** directory.

If you wish to keep the source tree unmodified (for example, for use in Section 5.13), please use the **--skip-patches** option.

5.8.2 dquilt and dpkg-source

The **quilt** command (or its wrapped **dquilt** command) was needed to manage the **-p1** patches in the **debian/patches/** directory before the **--commit** feature was added to the **dpkg-source** command in 1.16.1.

The patches should apply cleanly when using the **dpkg-source** command. Thus you can't just copy the patches to the new packaging of the new upstream release if there are patch offsets, etc.

The **dquilt** command (see Section 3.4) is more forgiving. You can normalize the patches by the **dquilt** command.

```
$ while dquilt push; do dquilt refresh ; done
$ dquilt pop -a
```

There is one advantage of using the **dpkg-source** command over the **dquilt** command. While the **dquilt** command cannot handle modified binary files automatically, the **dpkg-source** command detects modified binary files and lists them in the **debian/source/include-binaries** file to include them in the Debian tarball.

5.9 debian/upstream/signing-key.asc

Some packages are signed by a GPG key.

For example, GNU hello can be downloaded via HTTP from https://ftp.gnu.org/gnu/hello/ . There are sets of files:

- **hello**-*version*.**tar.gz** (upstream source)
- **hello**-*version*.**tar.gz.sig** (detached signature)

Let's pick the latest version set.

```
$ wget https://ftp.gnu.org/gnu/hello/hello-2.9.tar.gz
 ...
$ wget https://ftp.gnu.org/gnu/hello/hello-2.9.tar.gz.sig
 ...
$ gpg --verify hello-2.9.tar.gz.sig
gpg: Signature made Thu 10 Oct 2013 08:49:23 AM JST using DSA key ID 80EE4A00
gpg: Can't check signature: public key not found
```

If you know the public GPG key of the upstream maintainer from the mailing list, use it as the **debian/upstream/signing-key.asc** file. Otherwise, use the hkp keyserver and check it via your web of trust.

```
$ gpg --keyserver hkp://keys.gnupg.net --recv-key 80EE4A00
gpg: requesting key 80EE4A00 from hkp server keys.gnupg.net
gpg: key 80EE4A00: public key "Reuben Thomas <rrt@sc3d.org>" imported
gpg: no ultimately trusted keys found
gpg: Total number processed: 1
gpg:               imported: 1
$ gpg --verify hello-2.9.tar.gz.sig
gpg: Signature made Thu 10 Oct 2013 08:49:23 AM JST using DSA key ID 80EE4A00
gpg: Good signature from "Reuben Thomas <rrt@sc3d.org>"
  ...
Primary key fingerprint: 9297 8852 A62F A5E2 85B2  A174 6808 9F73 80EE 4A00
```

Tip

If your network environment blocks access to the HKP port **11371**, use "**hkp://keyserver.ubuntu.com:80**" instead.

After confirming the key ID **80EE4A00** is a trustworthy one, download its public key into the **debian/upstream/signing-key.asc** file.

```
$ gpg --armor --export 80EE4A00 >debian/upstream/signing-key.asc
```

Then set the corresponding **debian/watch** file as follows.

```
version=4
pgpsigurlmangle=s/$/.sig/  https://ftp.gnu.org/gnu/hello/ hello-(\d[\d.]*)\.tar ↩
    \.(?:gz|bz2|xz)
```

Now the **uscan** command will check the authenticity of the package using the GPG signature.

5.10 debian/watch and DFSG

Debian takes software freedom seriously and follows the DFSG.

The non-DFSG components in the upstream source tarball can be easily removed when the **uscan** command is used to update the Debian package.

- List the files to be removed in the **Files-Excluded** stanza of the **debian/copyright** file.
- List the URL to download the upstream tarball in the **debian/watch** file.
- Run the **uscan** command to download the new upstream tarball.
  - Alternatively, use the "**gbp import-orig --uscan --pristine-tar**" command.
- The resulting tarball has the version number with an additional suffix **+dfsg**.

5.11 Other debian/* Files

Optional configuration files may be added under the **debian/** directory. Most of them are to control **dh_*** commands offered by the **debhelper** package but there are some for **dpkg-source**, **lintian** and **gbp** commands.

> Tip
>
> 
>
> Check **debhelper**(7) for the latest available set of the **dh_*** commands.

These **debian/***binarypackage***.*** files provide very powerful means to set the installation path of files. Even an upstream source without its build system can be packaged just by using these files. See Section 8.2 as an example.

The "**-x[1234]**" superscript notation that appears in the following list indicates the minimum value for the **debmake -x** option that will generate the associated template file. See Section 6.6 or **debmake**(1) for details.

Here is the alphabetical list of notable optional configuration files.

binarypackage*.bug-control** **-x3** installed as **usr/share/bug/binarypackage*/**control** in *binarypackage*. See Section 5.24.

binarypackage*.bug-presubj** **-x3** installed as **usr/share/bug/binarypackage*/**presubj** in *binarypackage*. See Section 5.24.

binarypackage*.bug-script** **-x3** installed as **usr/share/bug/binarypackage* or **usr/share/bug/***binarypackage*/**script** in *binarypackage*. See Section 5.24.

***binarypackage*.bash-completion** List `bash` completion scripts to be installed.

The `bash-completion` package is required for both build and user environments.

See **dh_bash-completion**(1).

clean **-x2** List files that should be removed but are not cleaned by the **dh_auto_clean** command.

See **dh_auto_clean**(1) and **dh_clean**(1).

compat **-x1** Set the **debhelper** compatibility level (currently, "**9**").

See "COMPATIBILITY LEVELS" in **debhelper**(8).

***binarypackage*.conffile** No need for this file under "compat >= 3" since all files in the **etc/** directory are conffiles.

If the program you're packaging requires every user to modify the configuration files in the **/etc** directory, there are two popular ways to arrange for them not to be conffiles, keeping the **dpkg** command happy and quiet.

- Create a symlink under the **/etc** directory pointing to a file under the **/var** directory generated by the maintainer scripts.
- Create a file generated by the maintainer scripts under the **/etc** directory.

See **dh_installdeb**(1).

***binarypackage*.config** This is the **debconf config** script used for asking any questions necessary to configure the package. See Section 5.19.

***binarypackage*.cron.hourly** [-x3] Installed into the **etc/cron/hourly**/*binarypackage* file in *binarypackage*.

See **dh_installcron**(1) and **cron**(8).

***binarypackage*.cron.daily** [-x3] Installed into the **etc/cron/daily**/*binarypackage* file in *binarypackage*.

See **dh_installcron**(1) and **cron**(8).

***binarypackage*.cron.weekly** [-x3] Installed into the **etc/cron/weekly**/*binarypackage* file in *binarypackage*.

See **dh_installcron**(1) and **cron**(8).

***binarypackage*.cron.monthly** [-x3] Installed into the **etc/cron/monthly**/*binarypackage* file in *binarypackage*.

See **dh_installcron**(1) and **cron**(8).

binarypackage*.cron.d** [-x3] Installed into the **etc/cron.d/binarypackage* file in *binarypackage*.

See **dh_installcron**(1), **cron**(8), and **crontab**(5).

binarypackage*.default** [-x3] If this exists, it is installed into **etc/default/binarypackage* in *binarypackage*.

See **dh_installinit**(1).

***binarypackage*.dirs** [-x3] List directories to be created in *binarypackage*.

See **dh_installdirs**(1).

Usually, this is not needed since all **dh_install*** commands create required directories automatically. Use this only when you run into trouble.

***binarypackage*.doc-base** [-x2] Installed as the **doc-base** control file in *binarypackage*.

See **dh_installdocs**(1) and Debian doc-base Manual provided by the **doc-base** package.

***binarypackage*.docs** [-x2] List documentation files to be installed in *binarypackage*.

See **dh_installdocs**(1).

binarypackage*.emacsen-compat** [-x3] Installed into **usr/lib/emacsen-common/packages/compat/binarypackage* in *binarypackage*.

See **dh_installemacsen**(1).

binarypackage*.emacsen-install** [-x3] Installed into **usr/lib/emacsen-common/packages/install/binarypackage* in *binarypackage*.

See **dh_installemacsen**(1).

binarypackage*.emacsen-remove** [-x3] Installed into **usr/lib/emacsen-common/packages/remove/binarypackage* in *binarypackage*.

See **dh_installemacsen**(1).

binarypackage*.emacsen-startup** [-x3] Installed into **usr/lib/emacsen-common/packages/startup/binarypackage* in *binarypackage*.

See **dh_installemacsen**(1).

binarypackage*.examples** [-x2] List example files or directories to be installed into **usr/share/doc/binarypackage*/**examples/** in *binarypackage*.

See **dh_installexamples**(1).

gbp.conf If this exists, it functions as the configuration file for the **gbp** command.

See **gbp.conf**(5), **gbp**(1), and **git-buildpackage**(1).

***binarypackage*.info** [-x2] List info files to be installed in *binarypackage*.

See **dh_installinfo**(1).

binarypackage*.init** [-x3] Installed into **etc/init.d/binarypackage* in *binarypackage*.

See **dh_installinit**(1).

***binarypackage*.install** [-x2] List files which should be installed but are not installed by the **dh_auto_install** command.

See **dh_install**(1) and **dh_auto_install**(1).

license-examples/* [-x4] These are copyright file examples generated by the **debmake** command. Use these as the reference for making the **copyright** file.

Please make sure to erase these files.

***binarypackage*.links** [-x2] List pairs of source and destination files to be symlinked. Each pair should be put on its own line, with the source and destination separated by whitespace.

See **dh_link**(1).

binarypackage*.lintian-overrides** [-x3] Installed into **usr/share/lintian/overrides/binarypackage* in the package build directory. This file is used to suppress erroneous **lintian** diagnostics.

See **dh_lintian**(1), **lintian**(1) and Lintian User's Manual.

manpage.* [-x3] These are manpage template files generated by the **debmake** command. Please rename these to appropriate file names and update their contents.

Debian Policy requires that each program, utility, and function should have an associated manual page included in the same package. Manual pages are written in **nroff**(1).

If you are new to making a manpage, use **manpage.asciidoc** or **manpage.1** as the starting point.

***binarypackage*.manpages** [-x2] List man pages to be installed.

See **dh_installman**(1).

***binarypackage*.menu (deprecated, no more installed)** tech-ctte #741573 decided "Debian should use **.desktop** files as appropriate".

Debian menu file installed into **usr/share/menu/***binarypackage* in *binarypackage*.

See **menufile**(5) for its format. See **dh_installmenu**(1).

NEWS Installed into **usr/share/doc/***binarypackage*/**NEWS.Debian**.

See **dh_installchangelogs**(1).

patches/* Collection of **-p1** patch files which are applied to the upstream source before building the source.

See **dpkg-source**(1), Section 3.4 and Section 4.8.

No patch files are generated by the **debmake** command.

patches/series [-x1] The application sequence of the **patches/*** patch files.

***binarypackage*.preinst** [-x2] **, *binarypackage*.postinst** [-x2] **, *binarypackage*.prerm** [-x2] **, *binarypackage*.postrm** [-x2] These maintainer scripts are installed into the **DEBIAN** directory.

Inside the scripts, the token **#DEBHELPER#** is replaced with shell script snippets generated by other **debhelper** commands.

See **dh_installdeb**(1) and Chapter 6 - Package maintainer scripts and installation procedure in the "Debian Policy Manual".

See also **debconf-devel**(7) and 3.9.1 Prompting in maintainer scripts in the "Debian Policy Manual".

README.Debian -x1 Installed into the first binary package listed in the **debian/control** file as **usr/share/-doc**/*binarypackage*/**README.Debian**.

See **dh_installdocs**(1).

This file provides the information specific to the Debian package.

***binarypackage*.service** -x3 If this exists, it is installed into **lib/systemd/system**/*binarypackage*.**service** in *binarypackage*.

See **dh_systemd_enable**(1), **dh_systemd_start**(1), and **dh_installinit**(1).

source/format -x1 The Debian package format.

- Use "**3.0 (quilt)**" to make this non-native package (recommended)
- Use "**3.0 (native)**" to make this native package

See "SOURCE PACKAGE FORMATS" in **dpkg-source**(1).

source/lintian-overrides or source.lintian-overrides -x3 These files are not installed, but will be scanned by the **lintian** command to provide overrides for the source package.

See **dh_lintian**(1) and **lintian**(1).

source/local-options -x1 The **dpkg-source** command uses this content as its options. Notable options are:

- **unapply-patches**
- **abort-on-upstream-changes**
- **auto-commit**
- **single-debian-patch**

This is not included in the generated source package and is meant to be committed to the VCS of the maintainer.

See "FILE FORMATS" in **dpkg-source**(1).

source/local-patch-header Free form text that is put on top of the automatic patch generated.

This is not included in the generated source package and is meant to be committed to the VCS of the maintainer.

+ See "FILE FORMATS" in **dpkg-source**(1).

***binarypackage*.symbols** -x2 The symbols files, if present, are passed to the **dpkg-gensymbols** command to be processed and installed.

See **dh_makeshlibs**(1) and Section 5.18.1..

***binarypackage*.templates** This is the **debconf templates** file used for asking any questions necessary to configure the package. See Section 5.19.

tests/control This is the RFC822-style test meta data file defined in DEP-8. See **autopkgtest**(1) and Section 5.22.

TODO Installed into the first binary package listed in the **debian/control** file as **usr/share/doc**/*binarypackage*/**TODO.Debian**.

See **dh_installdocs**(1).

***binarypackage*.tmpfile** -x3 If this exists, it is installed into **usr/lib/tmpfiles.d**/*binarypackage*.**conf** in *binarypackage*.

See **dh_systemd_enable**(1), **dh_systemd_start**(1), and **dh_installinit**(1).

***binarypackage*.upstart** -x3 If this exists, it is installed into etc/init/package.conf in the package build directory. (deprecated)

See **dh_installinit**(1) and Section 8.1.

watch -x1 The control file for the **uscan** command to download the latest upstream version.

This control file may be configured to verify the authenticity of the tarball using its GPG signature (see Section 5.9).

See Section 5.10 and **uscan**(1).

Here are a few reminders for the above list.

- For a single binary package, the *binarypackage***.** part of the filename in the list may be removed.
- For a multi binary package, a configuration file missing the *binarypackage***.** part of the filename is applied to the first binary package listed in the **debian/control**.
- When there are many binary packages, their configurations can be specified independently by prefixing their name to their configuration filenames such as *package-1***.install**, *package-2***.install**, etc.
- Some template configuration files may not be created by the **debmake** command. In such cases, you need to create them with an editor.
- Unusual configuration template files generated by the **debmake** command with an extra **.ex** suffix need to be activated by removing that suffix.
- Unused configuration template files generated by the **debmake** command should be removed.
- Copy configuration template files as needed to the filenames matching their pertinent binary package names.

5.12 Customization of the Debian packaging

Let's recap the customization of the Debian packaging.

All customization data for the Debian package resides in the **debian/** directory. A simple example is given in Section 4.6. Normally, this customization involves a combination of the following:

- The Debian package build system can be customized through the **debian/rules** file (see Section 5.4.3).
- The Debian package installation path etc. can be customized through the addition of configuration files such as *package***.install** and *package***.docs** in the **debian/** directory for the **dh_*** commands from the **debhelper** package (see Section 5.11).

When these are not sufficient to make a good Debian package, modifications to the upstream source recorded as the **-p1** patches in the **debian/patches/** directory is deployed. These patches are applied in the sequence defined in the **debian/patches/series** file before building the package (see Section 5.8). Simple examples are given in Section 4.8.

You should address the root cause of the Debian packaging problem by the least invasive way. The generated package shall be more robust for future upgrades in this way.

Note

Send the patch addressing the root cause to the upstream maintainer if it is useful to the upstream.

5.13 Recording in VCS (standard)

Typically, Git is used as the VCS to record the Debian packaging activity with the following branches.

- **master** branch
  - Record the source tree used for the Debian packaging.
  - The upstream portion of the source tree is recorded unmodified.
  - The upstream modifications for the Debian packaging are recorded in the **debian/patches/** directory as the **-p1** patches.
- **upstream** branch
  - Record the upstream source tree untarred from the released upstream tarball.

Tip

It's a good idea to add to the **.gitignore** file the listing **.pc**.

Tip

Add **unapply-patches** and **abort-on-upstream-changes** lines to the **debian/source/local-options** file to keep the upstream portion unmodified.

Tip

You may also track the upstream VCS data with a branch different from the **upstream** branch to ease cherry-picking of patches.

5.14 Recording in VCS (alternative)

You may not wish to keep up with creating the **-p1** patch files for all upstream changes needed. You can record the Debian packaging activity with the following branches.

- **master** branch
  - Record the source tree used for the Debian packaging.
  - The upstream portion of the source tree is recorded with modifications for the Debian packaging.
- **upstream** branch
  - Record the upstream source tree untarred from the released upstream tarball.

Adding a few extra files in the **debian/** directory enables you to do this.

```
$ tar -xvzf <package-version>.tar.gz
$ ln -sf <package_version>.orig.tar.gz
$ cd <package-version>/
... hack...hack...
$ echo "single-debian-patch" >> debian/source/local-options
$ cat >debian/source/local-patch-header <<END
This patch contains all the Debian-specific changes mixed
together. To review them separately, please inspect the VCS
history at https://git.debian.org/?=collab-maint/foo.git.
```

Let the **dpkg-source** command invoked by the Debian package build process (**dpkg-buildpackage**, **debuild**, …) generate the **-p1** patch file **debian/patches/debian-changes** automatically.

Tip

This approach can be adopted for any VCS tools. Since this approach merges all changes into a merged patch, it is desirable to keep the VCS data publicly accessible.

Tip

The **debian/source/local-options** and **debian/source/local-patch-header** files are meant to be recorded in the VCS. These aren't included in the Debian source package.

5.15 Building package without extraneous contents

There are a few cases which cause the inclusion of undesirable contents in the generated Debian source package.

- The upstream source tree may be placed under the version control system. When the package is rebuilt from this source tree, the generated Debian source package contains extraneous contents from the version control system files.
- The upstream source tree may contain some auto-generated files. When the package is rebuilt from this source tree, the generated Debian source package contains extraneous contents from the auto-generated files.

Normally, the **-i** and **-I** options set in Section 3.5 for the **dpkg-source** command should avoid these. Here, the **-i** option is aimed at the non-native package while the **-I** is aimed at the native package. See **dpkg-source**(1) and the "**dpkg-source --help**" output.

There are several methods to avoid inclusion of undesirable contents.

5.15.1 Fix by debian/rules clean

The problem of extraneous contents can be fixed by removing such files in the "**debian/rules clean**" target. This is also useful for auto-generated files.

Note

The "**debian/rules clean**" target is called before the "**dpkg-source --build**" command by the **dpkg-buildpackage** command and the "**dpkg-source --build**" command ignores removed files.

5.15.2 Fix using VCS

The problem of extraneous contents can be fixed by restoring the source tree by committing the source tree to the VCS before the first build.

You can restore the source tree before the second package build. For example:

```
$ git reset --hard
$ git clean -dfx
$ debuild
```

This works because the **dpkg-source** command ignores the contents of the typical VCS files in the source tree with the **DEBUILD_DPKG_BUILDPACKAGE_OPTS** setting in Section 3.5.

Tip

If the source tree is not managed by a VCS, you should run "**git init; git add -A .; git commit**" before the first build.

5.15.3 Fix by extend-diff-ignore

This is for a non-native package.

The problem of extraneous diffs can be fixed by ignoring changes made to parts of the source tree by adding the "**extend-diff-ignore=…**" line in the **debian/source/options** file.

For excluding the **config.sub**, **config.guess** and **Makefile** files:

```
# Don't store changes on autogenerated files
extend-diff-ignore = "(^|/)(config\.sub|config\.guess|Makefile)$"
```

Note

This approach always works, even when you can't remove the file. So it saves you having to make a backup of the unmodified file just to be able to restore it before the next build.

Tip

If the **debian/source/local-options** file is used instead, you can hide this setting from the generated source package. This may be useful when the local non-standard VCS files interfere with your packaging.

5.15.4 Fix by tar-ignore

This is for a native package.

You can exclude some files in the source tree from the generated tarball by tweaking the file glob by adding the "**tar-ignore=…**" lines in the **debian/source/options** or **debian/source/local-options** files.

Note

If, for example, the source package of a native package needs files with the file extension **.o** as a part of the test data, the setting in Section 3.5 is too aggressive. You can work around this problem by dropping the **-I** option for **DEBUILD_DPKG_BUILDPACKAGE_OPTS** in Section 3.5 while adding the "**tar-ignore=…**" lines in the **debian/source/local-options** file for each package.

5.16 Upstream build systems

Upstream build systems are designed to go through several steps to install generated binary files to the system from the source distribution.

Tip

Before attempting to make a Debian package, you should become familiar with the upstream build system of the upstream source code and try to build it.

5.16.1 Autotools

Autotools (**autoconf** + **automake**) has 4 steps.

1. set up the build system ("**vim configure.ac Makefile.am**" and "**autoreconf -ivf**")
2. configure the build system ("**./configure**")
3. build the source tree ("**make**")
4. install the binary files ("**make install**")

The upstream maintainer usually performs step 1 and builds the upstream tarball for distribution using the "**make dist**" command. (The generated tarball contains not only the pristine upstream VCS contents but also other generated files.)

The package maintainer needs to take care of steps 2 to 4 at least. This is realized by the "**dh $@ --with autotools-dev**" command used in the **debian/rules** file.

The package maintainer may wish to take care all steps 1 to 4. This is realized by the "**dh $@ --with autoreconf**" command used in the **debian/rules** file. This rebuilds all auto-generated files to the latest version and provides better support for porting to the newer architectures.

For **compat** level **10** or newer, the simple "**dh $@**" command without "**--with autoreconf**" option can take care all steps 1 to 4, too.

If you wish to learn more on Autotools, please see:

- GNU Automake documentation
- GNU Autoconf documentation
- Autotools Tutorial
- Introduction to the autotools (autoconf, automake, and libtool)
- Autotools Mythbuster

5.16.2 CMake

CMake has 4 steps.

1. set up the build system ("**vim CMakeLists.txt config.h.in**")
2. configure the build system ("**cmake**")
3. build the source tree ("**make**")
4. install the binary files ("**make install**")

The upstream tarball contains no auto-generated files and is generated by the **tar** command after step 1.
The package maintainer needs to take care of steps 2 to 4.
If you wish to learn more on the CMake, please see:

- CMake
- CMake tutorial

5.16.3 Python distutils

Python distutils has 3 steps.

1. set up and configure the build system ("**vim setup.py**")
2. build the source tree ("**python setup.py build**")
3. install the binary files ("**python setup.py install**")

The upstream maintainer usually performs step 1 and builds the upstream tarball for distribution using the "**python setup.py sdist**" command.

The package maintainer needs to take care of step 2. This is realized simply by the "**dh $@**" command used in the **debian/rules** file, after **jessie**.

The situation of other build systems, such as CMake, are very similar to this Python one.

If you wish to learn more on Python3 and **distutils**, please see:

- Python3
- distutils

5.17 Debugging information

The Debian package is built with the debugging information but packaged into the binary package after stripping the debugging information as required by Chapter 10 - Files of the "Debian Policy Manual".

See

- 6.7.9. Best practices for debug packages of the "Debian Developer's Reference".
- 18.2 Debugging Information in Separate Files of the "Debugging with gdb"
- **dh_strip**(1)
- **strip**(1)
- **readelf**(1)
- **objcopy**(1)
- Debian wiki DebugPackage
- Debian wiki AutomaticDebugPackages
- Debian debian-devel post: Status on automatic debug packages (2015-08-15)

5.17.1 New -dbgsym package (Stretch 9.0 and after)

The debugging information is automatically packaged separately as the debug package using the **dh_strip** command with its default behavior. The name of such a debug package normally has the **-dbgsym** suffix.

If there were no **-dbg** packages defined in the **debian/control** file, no special care is needed for updating the package after the Stretch 9.0 release.

- The **debian/rules** file shouldn't explicitly contain **dh_strip**.
- Set **debian/compat** to **11** or newer.
- Bump the **Build-Depends** to **debhelper (>=11~)** or newer.

If there were **-dbg** packages defined in the **debian/control** file, following care is needed for updating the old package after the Stretch 9.0 release.

- Drop definition entries of such **-dbg** packages in the **debian/control** file.
- Replace "**dh_strip --dbg-package**=*package*" with "**dh_strip --dbgsym-migration**=*package*" in the **debian/rules** file to avoid file conflicts with the (now obsolete) **-dbg** package. See **dh_strip**(1).
- Set **debian/compat** to **11** or newer.
- Bump the **Build-Depends** to **debhelper (>=11~)** or newer.

5.18 Library package

Packaging library software requires you to perform much more work than usual. Here are some reminders for packaging library software:

- The library binary package must be named as in Section 5.5.1.3.
- Debian ships shared libraries such as **/usr/lib/<triplet>/lib***foo-0.1***.so.***1.0.0* (see Section 5.20).
- Debian encourages using versioned symbols in the shared library (see Section 5.18.1).
- Debian doesn't ship ***.la** libtool library archive files.
- Debian discourages using and shipping ***.a** static library files.

Before packaging shared library software, see:

- Chapter 8 - Shared libraries of the "Debian Policy Manual"
- 10.2 Libraries of the "Debian Policy Manual"
- 6.7.2. Libraries of the "Debian Developer's Reference"

For the historic background study, see:

- Escaping the Dependency Hell [6]
  - This encourages having versioned symbols in the shared library.
- Debian Library Packaging guide [7]
  - Please read the discussion thread following its announcement, too.

5.18.1 Library symbols

The symbols support in **dpkg** introduced in Debian **lenny** (5.0, May 2009) helps us to manage the backward ABI compatibility of the library package with the same package name. The **DEBIAN/symbols** file in the binary package provides the minimal version associated with each symbol.

An oversimplified method for the library packaging is as follows.

- Extract the old **DEBIAN/symbols** file of the immediate previous binary package with the "**dpkg-deb -e**" command.
  - Alternatively, the **mc** command may be used to extract the **DEBIAN/symbols** file.
- Copy it to the **debian/***binarypackage***.symbols** file.
  - If this is the first package, use an empty content file instead.
- Build the binary package.
  - If the **dpkg-gensymbols** command warns about some new symbols:
    - Extract the updated **DEBIAN/symbols** file with the "**dpkg-deb -e**" command.
    - Trim the Debian revision such as **-1** in it.
    - Copy it to the **debian/***binarypackage***.symbols** file.
    - Re-build the binary package.
  - If the **dpkg-gensymbols** command does not warn about new symbols:
    - You are done with the library packaging.

For the details, you should read the following primary references.

[6] This document was written before the introduction of the **symbols** file.

[7] The strong preference is to use the SONAME versioned **-dev** package names over the single **-dev** package name in Chapter 6. Development (-DEV) packages, which does not seem to be shared by the former ftp-master (Steve Langasek). This document was written before the introduction of the **multiarch** system and the **symbols** file.

- 8.6.3 The symbols system of the “Debian Policy Manual”
- **dh_makeshlibs**(1)
- **dpkg-gensymbols**(1)
- **dpkg-shlibdeps**(1)
- **deb-symbols**(5)

Yous should also check:

- Debian wiki UsingSymbolsFiles
- Debian wiki Projects/ImprovedDpkgShlibdeps
- Debian kde team Working with symbols files
- Section 8.11
- Section 8.12

Tip

For C++ libraries and other cases where the tracking of symbols is problematic, follow 8.6.4 The shlibs system of the “Debian Policy Manual”, instead. Please make sure to erase the empty **debian/***binarypackage***.symbols** file generated by the **debmake** command. For this case, the **DEBIAN/shlibs** file is used.

5.18.2 Library transition

When you package a new library package version which affects other packages, you must file a transition bug report against the **release.debian.org** pseudo package using the **reportbug** command with the ben file and wait for the approval for its upload from the Release Team.

Release team has the transition tracker. See Transitions.

Caution

Please make sure to rename binary packages as in Section 5.5.1.3.

5.19 debconf

The **debconf** package enables us to configure packages during their installation in 2 main ways:

- non-interactively from the **debian-installer** pre-seeding.
- interactively from the menu interface (**dialog**, **gnome**, **kde**, …)
  - the package installation: invoked by the **dpkg** command
  - the installed package: invoked by the **dpkg-reconfigure** command

All user interactions for the package installation must be handled by this **debconf** system using the following files.

- **debian/***binarypackage***.config**
  - This is the **debconf config** script used for asking any questions necessary to configure the package.

- **debian/***binarypackage***.template**
  - This is the **debconf templates** file used for asking any questions necessary to configure the package.
- package configuration scripts
  - **debian/***binarypackage***.preinst**
  - **debian/***binarypackage***.prerm**
  - **debian/***binarypackage***.postinst**
  - **debian/***binarypackage***.postrm**

See **dh_installdebconf**(1), **debconf**(7), **debconf-devel**(7) and 3.9.1 Prompting in maintainer scripts in the "Debian Policy Manual".

5.20 Multiarch

Multiarch support for cross-architecture installation of binary packages (particularly **i386** and **amd64**, but also other combinations) in the **dpkg** and **apt** packages introduced in Debian **wheezy** (7.0, May 2013), demands that we pay extra attention to packaging.

You should read the following references in detail.

- Ubuntu wiki (upstream)
  - MultiarchSpec
- Debian wiki (Debian situation)
  - Debian multiarch support
  - Multiarch/Implementation

The multiarch is enabled by using the **<triplet>** value such as **i386-linux-gnu** and **x86_64-linux-gnu** in the install path of shared libraries as **/usr/lib/<triplet>/**, etc..

- The **<triplet>** value required internally by **debhelper** scripts is implicitly set in themselves. The maintainer doesn't need to worry.
- The **<triplet>** value used in **override_dh_*** target scripts must be explicitly set in the **debian/rules** file by the maintainer. The **<triplet>** value is stored in the **$(DEB_HOST_MULTIARCH)** variable in the following **debian/rules** snippet example:

```
DEB_HOST_MULTIARCH = $(shell dpkg-architecture -qDEB_HOST_MULTIARCH)
...
override_dh_install:
        mkdir -p package1/lib/$(DEB_HOST_MULTIARCH)
        cp -dR tmp/lib/. package1/lib/$(DEB_HOST_MULTIARCH)
```

See:

- Section 5.4.4
- **dpkg-architecture**(1)
- Section 5.5.1.1
- Section 5.5.1.2

Table 5.1 The multiarch library path options

Classic path	i386 multiarch path	amd64 multiarch path
/lib/	**/lib/i386-linux-gnu/**	**/lib/x86_64-linux-gnu/**
/usr/lib/	**/usr/lib/i386-linux-gnu/**	**/usr/lib/x86_64-linux-gnu/**

5.20.1 The multiarch library path

Debian policy requires following Filesystem Hierarchy Standard. Its /usr/lib : Libraries for programming and packages states "**/usr/lib** includes object files, libraries, and internal binaries that are not intended to be executed directly by users or shell scripts."

Debian policy makes an exception to the Filesystem Hierarchy Standard to use **/usr/lib/<triplet>/** instead of **/usr/lib<qual>/** (e.g., **/lib32/** and **/lib64/**) to support a multiarch library.

For Autotools based packages under the **debhelper** package (compat>=9), this path setting is automatically taken care by the **dh_auto_configure** command.

For other packages with non-supported build systems, you need to manually adjust the install path as follows.

- If "**./configure**" is used in the **override_dh_auto_configure** target in **debian/rules**, make sure to replace it with "**dh_auto_configure** --" while re-targeting the install path from **/usr/lib/** to **/usr/lib/$(DEB_HOST_MULTIARCH)/**.
- Replace all occurrences of **/usr/lib/** with **/usr/lib/*/** in **debian/***foo***.install** files.

All files installed simultaneously as the multiarch package to the same file path should have exactly the same file content. You must be careful with differences generated by the data byte order and by the compression algorithm.

Note

The **--libexecdir** option of the **./configure** command specifies the default path to install executable programs run by other programs rather than by users. Its Autotools default is **/usr/libexec/** but its Debian non-multi-arch default is **/usr/lib/**. If such executables are a part of a "Multi-arch: foreign" package, a path such as **/usr/lib/** or **/usr/lib/***packagename* may be more desirable than **/usr/lib/<triplet>/**, which **dh_auto_configure** uses. The GNU Coding Standards: 7.2.5 Variables for Installation Directories has a description for **libexecdir** as "The definition of **libexecdir** is the same for all packages, so you should install your data in a subdirectory thereof. Most packages install their data under **$(libexecdir)/package-name/** …". (It is always a good idea to follow GNU unless it conflicts with the Debian policy.)

The shared library files in the default path **/usr/lib/** and **/usr/lib/<triplet>/** are loaded automatically.

For shared library files in another path, the GCC option **-l** must be set by the **pkg-config** command to make them load properly.

5.20.2 The multiarch header file path

GCC includes both **/usr/include/** and **/usr/include/<triplet>/** by default on the multiarch Debian system.

If the header file is not in those paths, the GCC option **-I** must be set by the **pkg-config** command to make "**#include** <*foo.h*>" work properly.

Table 5.2 The multiarch header file path options

Classic path	i386 multiarch path	amd64 multiarch path
/usr/include/	**/usr/include/i386-linux-gnu/**	**/usr/include/x86_64-linux-gnu/**
/usr/include/*packagename***/**	**/usr/include/i386-linux-gnu/***packagename***/**	**/usr/include/x86_64-linux-gnu/***packagename***/**
	/usr/lib/i386-linux-gnu/*packagename***/**	**/usr/lib/x86_64-linux-gnu/***packagename***/**

The use of the **/usr/lib/<triplet>**/*packagename*/ path for the library files allows the upstream maintainer to use the same install script for the multiatch system with **/usr/lib/<triplet>** and the biarch system with **/usr/lib<qual>**/. [8]

The use of the file path containing *packagename* enables having more than 2 development libraries simultaneously installed on a system.

5.20.3 The multiarch *.pc file path

The **pkg-config** program is used to retrieve information about installed libraries in the system. It stores its configuration parameters in the ***.pc** file and is used for setting the **-I** and **-l** options for GCC.

Table 5.3 The ***.pc** file path options

Classic path	i386 multiarch path	amd64 multiarch path
/usr/lib/pkgconfig/	**/usr/lib/i386-linux-gnu/pkgconfig/**	**/usr/lib/x86_64-linux-gnu/pkgconfig/**

5.21 Compiler hardening

The compiler hardening support spreading for Debian **jessie** (8.0, TBA) demands that we pay extra attention to the packaging.

You should read the following references in detail.

- Debian wiki Hardening
- Debian wiki Hardening Walkthrough

The **debmake** command adds template comments to the **debian/rules** file as needed for **DEB_BUILD_MAINT_OPTIONS**, **DEB_CFLAGS_MAINT_APPEND**, and **DEB_LDFLAGS_MAINT_APPEND** (see Chapter 4 and **dpkg-buildflags**(1)).

5.22 Continuous integration

DEP-8 defines the **debian/tests/control** file as the RFC822-style test metadata file for continuous integration (CI) of the Debian package.

It is used after building the binary packages from the source package containing this **debian/tests/control** file. When the **autopkgtest** command is run, the generated binary packages are installed and tested in the virtual environment according to this file.

See documents in the **/usr/share/doc/autopkgtest/** directory and 3. autopkgtest: Automatic testing for packages of the "Ubuntu Packaging Guide".

There are several other CI tools on Debian for you to explore.

- The **debci** package: CI platform on top of the **autopkgtest** package
- The **jenkins** package: generic CI platform

5.23 Bootstrapping

Debian cares about supporting new ports or flavours. The new ports or flavours require bootstrapping operation for the cross-build of the initial minimal native-building system. In order to avoid build-dependency loops during bootstrapping, the build-dependency needs to be reduced using the profile builds feature.

[8] This path is compliant with the FHS. Filesystem Hierarchy Standard: /usr/lib : Libraries for programming and packages states "Applications may use a single subdirectory under **/usr/lib**. If an application uses a subdirectory, all architecture-dependent data exclusively used by the application must be placed within that subdirectory."

Tip

If a core package `foo` build depends on a package `bar` with deep build dependency chains but `bar` is only used in the **test** target in `foo`, you can safely mark the `bar` with **<!nocheck>** in the **Build-depends** of `foo` to avoid build loops.

5.24 Bug reports

The **reportbug** command used for the bug report of *binarypackage* can be customized by the files in **usr/share/bug/***binarypackage*
The **dh_bugfiles** command installs these files from the template files in the **debian/** directory.

- **debian/***binarypackage***.bug-control** → **usr/share/bug/***binarypackage***/control**
  - This file contains some directions such as redirecting the bug report to another package.
- **debian/***binarypackage***.bug-presubj** → **usr/share/bug/***binarypackage***/presubj**
  - This file is displayed to the user by the **reportbug** command.
- **debian/***binarypackage***.bug-script** → **usr/share/bug/***binarypackage* or **usr/share/bug/***binarypackage***/script**
  - The **reportbug** command runs this script to generate a template file for the bug report.

See **dh_bugfiles**(1) and reportbug's Features for Developers

Tip

If you always remind the bug reporter of something or ask them about their situation, use these files to automate it.

Chapter 6

debmake options

Here are some notable options for the **debmake** command.

6.1 Shortcut options (-a, -i)

The **debmake** command offers 2 shortcut options.

- **-a** : open the upstream tarball
- **-i** : execute script to build the binary package

The example in the above Chapter 4 can be done simply as follows.

```
$ debmake -a package-1.0.tar.gz -i debuild
```

Tip

A URL such as "**https://www.example.org/DL/package-1.0.tar.gz**" may be used for the **-a** option.

Tip

A URL such as "**https://arm.koji.fedoraproject.org/packages/ibus/1.5.7/3.fc21/src/ibus-1.5.7-3.fc21.src.rpm**" may be used for the **-a** option, too.

6.1.1 Python module

You can generate a functioning single binary Debian package with a reasonable package description directly from the Python module package offered as a tarball, *pythonmodule-1.0***.tar.gz**. The **-b** option specifying the package type **python** and the **-s** option to copy the package description from the upstream package need to be specified.

```
$ debmake -s -b':python' -a pythonmodule-1.0.tar.gz -i debuild
```

For other interpreted languages that support the **-b** option, specify the pertinent *type* for the **-b** option.

For interpreted languages without the **-b** option support, specify the **script** type instead and add the interpreter package as a dependency of the resulting binary package by adjusting the **debian/control** file.

6.2 Upstream snapshot (-d, -t)

The upstream snapshot from the upstream source tree in the VCS can be made with the **-d** option if the upstream package supports the "**make dist**" equivalence.

```
$ cd /path/to/upstream-vcs
$ debmake -d -i debuild
```

Alternatively, the same can be made with the **-t** option if the upstream tarball can be made with the **tar** command.

```
$ cd /path/to/upstream-vcs
$ debmake -p package -t -i debuild
```

Unless you provide the upstream version with the **-u** option or with the **debian/changelog** file, a snapshot upstream version is generated in the **0~%y%m%d%H%M** format, e.g., *0~1403012359*, from the UTC date and time.

If the upstream VCS is hosted in the *package/* directory instead of the *upstream-vcs/* directory, the "**-p** *package*" can be skipped.

If the upstream source tree in the VCS contains the **debian/*** files, the **debmake** command with either the **-d** option or the **-t** option combined with the **-i** option automates the making of a non-native Debian package from the VCS snapshot while using these **debian/*** files.

```
$ cp -r /path/to/package-0~1403012359/debian/. /path/to/upstream-vcs/debian
$ dch
  ... update debian/changelog
$ git add -A .; git commit -m "vcs with debian/*"
$ debmake -t -p package -i debuild
```

This **non-native** Debian binary package building scheme using the "**debmake -t -i debuild**" command may be considered as the **quasi-native** Debian package scheme since the packaging situation resembles the **native** Debian binary package building case using the **debuild** command without the upstream tarball.

Use of a **non-native** Debian package helps to ease communication with the downstream distros such as Ubuntu for bug fixes etc.

6.3 debmake -cc

The **debmake** command with the **-cc** option can make a summary of the copyright and license for the entire source tree to standard output.

```
$ tar -xvzf package-1.0.tar.gz
$ cd package-1.0
$ debmake -cc | less
```

With the **-c** option, this provides shorter report.

6.4 debmake -k

When updating a package for the new upstream release, the **debmake** command can verify the content of the existing **debian/copyright** file against the copyright and license situation of the entire updated source tree.

```
$ cd package-vcs
$ gbp import-orig --uscan --pristine-tar
... update source with the new upstream release
$ debmake -k | less
```

The "**debmake -k**" command parses the **debian/copyright** file from the top to the bottom and compares the license of all the non-binary files in the current package with the license described in the last matching file pattern entry of the **debian/copyright** file.

When editing the auto-generated **debian/copyright** file, please make sure to keep the generic file patterns at the top of the list.

Tip

For all new upstream releases, run the "**debmake -k**" command to ensure that the **debian/copyright** file is current.

6.5 debmake -j

The generation of a functioning multi-binary package always requires more manual work than that of a functioning single binary package. The test build of the source package is the essential part of it.

For example, let's package the same *package-1.0***.tar.gz** (see Chapter 4) into a multi binary package.

- Invoke the **debmake** command with the **-j** option for the test building and the report generation.

```
$ debmake -j -a package-1.0.tar.gz
```

- Check the last lines of the *package***.build-dep.log** file to judge build dependencies for **Build-Depends**. (You do not need to list packages used by **debhelper**, **perl**, or **fakeroot** explicitly in **Build-Depends**. This technique is useful for the generation of a single binary package, too.)
- Check the contents of the *package***.install.log** file to identify the install paths for files to decide how you split them into multiple packages.
- Start packaging with the **debmake** command.

```
$ rm -rf package-1.0
$ tar -xvzf package-1.0.tar.gz
$ cd package-1.0
$ debmake -b"package1:type1, ..."
```

- Update **debian/control** and **debian/***binarypackage***.install** files using the above information.
- Update other **debian/*** files as needed.
- Build the Debian package with the **debuild** command or its equivalent.

```
$ debuild
```

- All binary package entries specified in the **debian/***binarypackage***.install** file are generated as *binarypackage_version-revision_arch***.deb**.

Note

The **-j** option for the **debmake** command invokes **dpkg-depcheck**(1) to run **debian/rules** under **strace**(1) to obtain library dependencies. Unfortunately, this is very slow. If you know the library package dependencies from other sources such as the SPEC file in the source, you may just run the "**debmake** …" command without the **-j** option and run the "**debian/rules install**" command to check the install paths of the generated files.

6.6 debmake -x

The amount of template files generated by the **debmake** command depends on the **-x**[01234] option.

- See Section 8.1 for cherry-picking of the template files.

Note

None of the existing configuration files are modified by the **debmake** command.

6.7 debmake -P

The **debmake** command invoked with the **-P** option pedantically checks auto-generated files for copyright+license text even if they are with permissive license.

This option affects not only the content of the **debian/copyright** file generated by normal execution, but also the output by the execution with the **-k**, **-c**, **-cc**, and **-ccc** options.

6.8 debmake -T

The **debmake** command invoked with the **-T** option additionally prints verbose tutorial comment lines. The lines marked with ### in the template files are part of the verbose tutorial comment lines.

Chapter 7

Tips

Here are some notable tips about Debian packaging.

7.1 debdiff

You can compare file contents in two source Debian packages with the **debdiff** command.

```
$ debdiff old-package.dsc new-package.dsc
```

You can also compare file lists in two sets of binary Debian packages with the **debdiff** command.

```
$ debdiff old-package.changes new-package.changes
```

These are useful to identify what has been changed in the source packages and to check for inadvertent changes made when updating binary packages, such as unintentionally misplacing or removing files.

7.2 dget

You can download the set of files for the Debian source package with the **dget** command.

```
$ dget https://www.example.org/path/to/package_version-rev.dsc
```

7.3 debc

You should install generated packages with the **debc** command to test it locally.

```
$ debc package_version-rev_arch.changes
```

7.4 piuparts

You should install generated packages with the **piuparts** command to test it automatically.

```
$ sudo piuparts package_version-rev_arch.changes
```

Note

This is a very slow process with remote APT package repository access.

7.5 debsign

After completing the test of the package, you can sign it with the **debsign** command.

```
$ debsign package_version-rev_arch.changes
```

7.6 dput

After signing the package with the **debsign** command, you can upload the set of files for the Debian source and binary packages with the **dput** command.

```
$ dput package_version-rev_arch.changes
```

7.7 bts

After uploading the package, you will receive bug reports. It is an important duty of a package maintainer to manage these bugs properly as described in 5.8. Handling bugs of the "Debian Developer's Reference".

The **bts** command is a handy tool to manage bugs on the Debian Bug Tracking System.

```
$ bts severity 123123 wishlist , tags -1 pending
```

7.8 git-buildpackage

The **git-buildpackage** package offers many commands to automate packaging activities using the git repository.

- **gbp import-dsc**: import the previous Debian source package to the git repository.
- **gbp import-orig**: import the new upstream tar to the git repository.
  - The **--pristine-tar** option for the **git import-orig** command enables storing the upstream tarball in the same git repository.
  - The **--uscan** option as the last argument of the **gbp import-orig** command enables downloading and committing the new upstream tarball into the git repository.
- **gbp dch**: generate the Debian changelog from the git commit messages.
- **gbp buildpackage**: build the Debian binary package from the git repository.
- **gbp pull**: update the **debian**, **upstream** and **pristine-tar** branches safely from the remote repository.
- **git-pbuilder**: build the Debian binary package from the git repository using the **pbuilder** package.
  - The **cowbuilder** package is used as its backend.
- The **gbp pq**, **git-dpm** or **quilt** (or alias **dquilt**) commands are used to manage quilt patches.

- The **dquilt** command is the simplest to learn and requires you to commit the resulting files manually with the **git** command to the **master** branch.
- The "**gbp pq**" command provides the equivalent functionality of patch set management without using **dquilt** and eases including upstream git repository changes by cherry-picking.
- The "**git dpm**" command provides more enhanced functionality than that of the '**gbp pq**'' command.

Package history management with the **git-buildpackage** package is becoming the standard practice for most Debian maintainers.

See:

- Building Debian Packages with git-buildpackage
- https://wiki.debian.org/GitPackagingWorkflow
- https://wiki.debian.org/GitPackagingWorkflow/DebConf11BOF
- https://raphaelhertzog.com/2010/11/18/4-tips-to-maintain-a-3-0-quilt-debian-source-package-in-a-vcs/
- The **systemd** packaging practice documentation on Building from source.

Tip

Relax. You don't need to use all the wrapper tools. Use only ones which match your needs.

7.8.1 gbp import-dscs --debsnap

For Debian source packages named <*source-package*> recorded in the snapshot.debian.org archive, an initial git repository with all of the Debian version history can be generated as follows.

```
$ gbp import-dscs --debsnap --pristine-tar '<source-package>'
```

7.9 Upstream git repository

For Debian packaging with the **git-buildpackage** package, the **upstream** branch on the remote repository **origin** is normally used to track the content of the released upstream tarball.

The upstream git repository can also be tracked by naming its remote repository as **upstream** instead of the default **origin**. Then you can easily cherry-pick recent upstream changes into the Debian revision by cherry-picking with the **gitk** command and using the **gbp-pq** command.

Tip

The "**gbp import-orig --upstream-vcs-tag**" command can create a nice packaging history by making a merge commit into the **upstream** branch from the specified tag on the upstream git repository.

Caution

The content of the released upstream tarball may not match exactly with the corresponding content of the upstream git repository. It may contain some auto-generated files or miss some files. (Autotools, distutils, …)

7.10 chroot

The chroot for a clean package build environment can be created and managed using the tools described in Chapter 3. [1]

Here is a quick summary of available package build commands. There are many ways to do the same thing.

- **dpkg-buildpackage** = core of package building tool
- **debuild** = **dpkg-buildpackage** + **lintian** (build under the sanitized environment variables)
- **pbuilder** = core of the Debian chroot environment tool
- **pdebuild** = **pbuilder** + **dpkg-buildpackage** (build in the chroot)
- **cowbuilder** = speed up the **pbuilder** execution
- **git-pbuilder** = the easy-to-use command line syntax for **pdebuild** (used by **gbp buildpackge**)
- **gbp** = manage the Debian source under git
- **gbp buildpackge** = **pbuilder** + **dpkg-buildpackage** + **gbp**

A clean **sid** distribution chroot environment can be used as follows.

- The chroot filesystem creation command for the **sid** distribution
  - **pbuilder create**
  - **git-pbuilder create**
- The master chroot filesystem path for the **sid** distribution chroot filesystem
  - **/var/cache/pbuilder/base.cow**
- The package build command for the **sid** distribution chroot
  - **pdebuild**
  - **git-pbuilder**
  - **gbp buildpackage**
- The command to update the **sid** chroot
  - **pbuilder --update**
  - **git-pbuilder update**
- The command to login to the **sid** chroot filesystem to modify it
  - **git-pbuilder login --save-after-login**

An arbitrary *dist* distribution environment can be used as follows.

- The chroot filesystem creation command for the *dist* distribution
  - **pbuilder create --distribution** *dist*
  - **DIST=***dist* **git-pbuilder create**
- The master chroot filesystem path for the *dist* distribution chroot
  - path: **/var/cache/pbuilder/base-***dist***.cow**
- The package build command for the *dist* distribution chroot
  - **pdebuild -- --basepath=/var/cache/pbuilder/base-***dist***.cow**

[1] The **git-pbuilder** style organization is deployed here. See https://wiki.debian.org/git-pbuilder . Be careful since many HOWTOs use different organization.

  - **DIST**=*dist* **git-pbuilder**
  - **gbp buildpackage --git-dist**=*dist*

- The command to update the *dist* chroot
  - **pbuilder update --basepath=/var/cache/pbuilder/base-*dist*.cow**
  - **DIST**=*dist* **git-pbuilder update**
- The command to login to the **dist** chroot to modify it
  - **pbuilder --login --basepath=/var/cache/pbuilder/base-*dist*.cow --save-after-login**
  - **DIST**=*dist* **git-pbuilder login --save-after-login**

Tip

A custom environment with some pre-loaded packages needed for the new experimental packages, this "**git-pbuilder login --save-after-login**" command is quite handy.

Tip

If your old chroot filesystem is missing packages such as **libeatmydata1**, **ccache**, and **lintian**, you may want to install these with the "**git-pbuilder login --save-after-login**" command.

Tip

The chroot filesystem can be cloned simply by copying with the "**cp -a base-*dist*.cow base-*customdist*.cow**" command. The new chroot filesystem can be accessed as "**gbp buildpackage --git-dist=***customdist*" and "**DIST=***customdist* **git-pbuilder …**".

Tip

When the **orig.tar.gz** file needs to be uploaded for a Debian revision other than **0** or **1** (e.g., for a security upload), add the **-sa** option to the end of **dpkg-buildpackage**, **debuild**, **pdebuild**, and **git-pbuilder** commands. For the "**gbp buildpackage**" command, temporarily modify the **builder** setting of **~/.gbp.conf**.

Note

The description in this section is too terse to be useful for most of the prospective maintainers. This is the intentional choice of the author. You are highly encouraged to search and read all the pertinent documents associated with the commands used.

7.11 New Debian revision

Let's assume that a bug report **#***bug_number* was filed against your package, and it describes a problem that you can solve by editing the **buggy** file in the upstream source. Here's what you need to do to create a new Debian revision of the package with the **bugname.patch** file recording the fix.

New Debian revision with the dquilt command

```
$ dquilt push -a
$ dquilt new bugname.patch
$ dquilt add buggy
$ vim buggy
  ...
$ dquilt refresh
$ dquilt header -e
$ dquilt pop -a
$ dch -i
```

Alternatively if the package is managed in the git repository using the **git-buildpackage** command with its default configuration:

New Debian revision with the gbp-pq command

```
$ git checkout master
$ gbp pq import
$ vim buggy
$ git add buggy
$ git commit
$ git tag pq/<newrev>
$ gbp pq export
$ gbp drop
$ git add debian/patches/*
$ dch -i
$ git commit -a -m "Closes: #<bug_number>"
```

Please make sure to describe concisely the changes that fix reported bugs and close those bugs by adding "**Closes: #***<bug_number>*" in the **debian/changelog** file.

Tip

Use a **debian/changelog** entry with a version string such as **1.0.1-1~rc1** when you experiment. Then, unclutter such **changelog** entries into a single entry for the official package.

7.12 New upstream release

If a package **foo** is properly packaged in the modern "**3.0 (native)**" or "**3.0 (quilt)**" formats, packaging a new upstream release is essentially moving the old **debian/** directory to the new source. This can be done by running the "**tar -xvzf** */path/to/foo_oldversion***.debian.tar.gz**" command in the new extracted source. [2] Of course, you need to do some obvious chores.

There are several tools to handle this situation. After updating to the new upstream release with these tools, please make sure to describe concisely the changes in the new upstream release that fix reported bugs and close those bugs by adding "**Closes: #***bug_number*" in the **debian/changelog** file.

7.12.1 uupdate + tarball

You can automatically update to the new upstream source with the **uupdate** command from the **devscripts** package. It requires having the old Debian source package and the new upstream tarball.

[2] If a package **foo** is packaged in the old **1.0** format, this can be done by running the "**zcat** */path/to/foo_oldversion***.diff.gz|patch -p1**" command in the new extracted source, instead.

```
$ wget https://example.org/foo/foo-newversion.tar.gz
$ cd foo-oldversion
$ uupdate -v newversion ../foo-newversion.tar.gz
...
$ cd ../foo-newversion
$ while dquilt push; do dquilt refresh; done
$ dch
```

7.12.2 uscan

You can automatically update to the new upstream source with the **uscan** command from the **devscripts** package. It requires having the old Debian source package and the **debian/watch** file in it.

```
$ cd foo-oldversion
$ uscan
...
$ while dquilt push; do dquilt refresh; done
$ dch
```

7.12.3 gbp

You can automatically update to the new upstream source with the "**gbp import-orig** --**pristine-tar**" command from the **git-buildpackage** package. It requires having the old Debian source in the git repository and the new upstream tarball.

```
$ ln -sf foo-newversion.tar.gz foo_newversion.orig.tar.gz
$ cd foo-vcs
$ git checkout master
$ gbp pq import
$ git checkout master
$ gbp import-orig --pristine-tar  ../foo_newversion.orig.tar.gz
...
$ gbp pq rebase
$ git checkout master
$ gbp pq export
$ gbp pq drop
$ git add debian/patches
$ dch -v <newversion>
$ git commit -a -m "Refresh patches"
```

Tip

If upstream uses a git repository, please also use the **--upstream-vcs-tag** option for the **gbp import-orig** command.

7.12.4 gbp + uscan

You can automatically update to the new upstream source with the "**gbp import-orig --pristine-tar --uscan**" command from the **git-buildpackage** package. It requires having the old Debian source in the git repository and the **debian/watch** file in it.

```
$ cd foo-vcs
$ git checkout master
$ gbp pq import
$ git checkout master
$ gbp import-orig --pristine-tar --uscan
...
$ gbp pq rebase
$ git checkout master
$ gbp pq export
$ gbp pq drop
$ git add debian/patches
$ dch -v <newversion>
$ git commit -a -m "Refresh patches"
```

Tip

If upstream uses a git repository, please also use the **--upstream-vcs-tag** option for the **gbp import-orig** command.

7.13 3.0 source format

Updating the package style is not a required activity for the update of a package. However, doing so lets you use the full capabilities of the modern **debhelper** system and the **3.0** source format.

- If you need to recreate deleted template files for any reason, you can run **debmake** again in the same Debian package source tree. Then edit them appropriately.
- If the package has not been updated to use the **dh** command for the **debian/rules** file, update it to use it (see Section 5.4.2). Update the **debian/control** file accordingly.
- If you have a **1.0** source package with the **foo.diff.gz** file, you can update it to the newer "**3.0 (quilt)**" source format by creating **debian/source/format** with "**3.0 (quilt)**". The rest of the **debian/*** files can just be copied. Import the **big.diff** file generated by the "**filterdiff -z -x *debian/* foo.diff.gz > big.diff**" command to your quilt system, if needed. [3]
- If it was packaged using another patch system such as **dpatch**, **dbs**, or **cdbs** with **-p0**, **-p1**, or **-p2**, convert it to the quilt command using the **deb3** script in the **quilt** package.
- If it was packaged with the **dh** command with the "**--with quilt**" option or with the **dh_quilt_patch** and **dh_quilt_unpatch** commands, remove these and make it use the newer "**3.0 (quilt)**" source format.
- If you have a **1.0** source package without the **foo.diff.gz** file, you can update it to the newer "**3.0 (native)**" source format by creating **debian/source/format** with "**3.0 (native)**". The rest of the **debian/*** files can just be copied.

You should check DEP - Debian Enhancement Proposals and adopt ACCEPTED proposals.
See ProjectsDebSrc3.0 to check the support status of the new Debian source formats by the Debian tool chains.

7.14 CDBS

The Common Debian Build System (**CDBS**) is a wrapper system over the **debhelper** package. The **CDBS** is based on the Makefile inclusion mechanism and configured by the **DEB_*** configuration variables set in the **debian/rules** file.

Before the introduction of the **dh** command to the **debhelper** package at the version 7, the **CDBS** was the only approach to create a simple and clean **debian/rules** file.

[3] You can split the **big.diff** file into many small incremental patch files using the **splitdiff** command.

For many simple packages, the **dh** command alone allows us to make a simple and clean **debian/rules** file now. It is desirable to keep the build system simple and clean by not using the superfluous **CDBS**.

Note

Neither “the **CDBS** magically does the job for me with less typing” nor “I don’t understand the new **dh** syntax” can be an excuse to keep using the **CDBS** system.

For some complicated packages such as GNOME related ones, the **CDBS** is leveraged to automate their uniform packaging by the current maintainers with justification. If this is the case, please do not bother converting from the **CDBS** to the **dh** syntax.

Note

If you are working with a team of maintainers, please follow the established practice of the team.

When converting packages from the **CDBS** to the **dh** syntax, please use the following as your reference:

- CDBS Documentation
- The Common Debian Build System (CDBS), FOSDEM 2009

7.15 Build under UTF-8

The default locale of the build environment is **C**.

Some programs such as the **read** function of Python3 change their behavior depending on the locale.

Adding the following code to the **debian/rules** file ensures building the program under the **C.UTF-8** locale.

```
LC_ALL := C.UTF-8
export LC_ALL
```

7.16 UTF-8 conversion

If upstream documents are encoded in old encoding schemes, converting them to UTF-8 is a good idea.

Use the **iconv** command in the **libc-bin** package to convert encodings of plain text files.

```
$ iconv -f latin1 -t utf8 foo_in.txt > foo_out.txt
```

Use **w3m**(1) to convert from HTML files to UTF-8 plain text files. When you do this, make sure to execute it under UTF-8 locale.

```
$ LC_ALL=C.UTF-8 w3m -o display_charset=UTF-8 \
       -cols 70 -dump -no-graph -T text/html \
       < foo_in.html > foo_out.txt
```

Run these scripts in the **override_dh_*** target of the **debian/rules** file.

7.17 Upload orig.tar.gz

When you first upload the package to the archive, you need to include the original **orig.tar.gz** source, too.

If the Debian revision number of the package is either **1** or **0**, this is the default. Otherwise, you must provide the **dpkg-buildpackage** option **-sa** to the **dpkg-buildpackage** command.

- **dpkg-buildpackage -sa**
- **debuild -sa**
- **pdebuild --debbuildopts -sa**
- **git-pbuilder -sa**
- For **gbp buildpackage**, edit the **~/.gbp.conf** file.

Tip

On the other hand, the **-sd** option will force the exclusion of the original **orig.tar.gz** source.

Tip

Security uploads require including the **orig.tar.gz** file.

7.18 Skipped uploads

If you created multiple entries in the **debian/changelog** while skipping uploads, you must create a proper ***_.changes** file which includes all changes since the last upload. This can be done by specifying the dpkg-buildpackage option **-v** with the last uploaded version, e.g., *1.2*.

- **dpkg-buildpackage -v***1.2*
- **debuild -v***1.2*
- **pdebuild --debbuildopts -v***1.2*
- **git-pbuilder -v***1.2*
- For **gbp buildpackage**, edit the **~/.gbp.conf** file.

7.19 Advanced packaging

Hints for the following can be found in the **debhelper**(7) manpage:

- differences of the **debhelper** tool behavior under "**compat** <= 8"
- building several binary packages with several different build conditions
  - making multiple copies of the upstream source
  - invoking multiple "**dh_auto_configure -S …**" commands in the **override_dh_auto_configure** target
  - invoking multiple "**dh_auto_build -S …**" commands in the **override_dh_auto_build** target
  - invoking multiple "**dh_auto_install -S …**" commands in the **override_dh_auto_install** target
- building **udeb** packages with "**Package-Type: udeb**" in **debian/control** (see Package-Type)
- excluding some packages for the bootstrapping process (see also BuildProfileSpec)
  - adding the **Build-Profiles** fields in binary package stanzas in **debian/control**

  - building packages with the **DEB_BUILD_PROFILES** environment variable set to the pertinent profile name

Hints for the following can be found in the **dpkg-source**(1) manpage:

- naming convention for multiple upstream source tarballs
  - *packagename_version*.**orig.tar.gz**
  - *packagename_version*.**orig-***componentname*.**tar.gz**
- recording the Debian changes to the upstream source package
  - **dpkg-source --commit**

7.20 Other distros

Although the upstream tarball has all the information to build the Debian package, it is not always easy to figure out which combination of options to use.

Also, the upstream package may be more focused on feature enhancements and may be less eager about backward compatibilities etc., which are an important aspect of Debian packaging practice.

The leveraging of information from other distributions is an option to address the above issues.

If the other distribution of interest is a Debian derivative one, it is trivial to reuse it.

If the other distribution of interest is an RPM based distribution, see Repackage src.rpm.

Downloading and opening of the **src.rpm** file can be done with the **rget** command. (Place the **rget** script in your PATH.)

rget script

```
#!/bin/sh
FCSRPM=$(basename $1)
mkdir ${FCSRPM}; cd ${FCSRPM}/
wget $1
rpm2cpio ${FCSRPM} | cpio -dium
```

Many upstream tarballs contain the SPEC file named as *packagename*.**spec** or *packagename*.**spec.in** used by the RPM system. This can be used as the baseline for the Debian package, too.

7.21 Debug

When you face build problems or core dumps of generated binary programs, you need to resolve them yourself. That's **debug**.

This is too deep a topic to describe here. So, let me just list few pointers and hints for some typical debug tools.

- core dump
  - "**man core**"
  - Update the "**/etc/security/limits.conf**" file to include the following:

```
* soft core unlimited
```

  - "**ulimit -c unlimited**" in **~/.bashrc**
  - "**ulimit -a**" to check
  - Press **Ctrl-**\ or "**kill -ABRT** *PID*" to make a core dump file
- **gdb** - The GNU Debugger
  - "**info gdb**"
  - "Debugging with GDB" in **/usr/share/doc/gdb-doc/html/gdb/index.html**

- **strace** - Trace system calls and signals
  - Use **strace-graph** script found in **/usr/share/doc/strace/examples/** to make a nice tree view
  - "**man strace**"
- **ltrace** - Trace library calls
  - "**man ltrace**"
- "**sh -n** *script.sh*" - Syntax check of a Shell script
- "**sh -x** *script.sh*" - Trace a Shell script
- "**python -m py_compile** *script.py*" - Syntax check of a Python script
- "**python -mtrace --trace** *script.py*" - Trace a Python script
- "**perl -I ../libpath -c** *script.pl*" - Syntax check of a Perl script
- "**perl -d:Trace** *script.pl*" - Trace a Perl script
  - Install the **libterm-readline-gnu-perl** package or its equivalent to add input line editing capability with history support.
- **lsof** - List open files by processes
  - "**man lsof**"

Tip

The **script** command records console outputs.

Tip

The **screen** and **tmux** commands used with the **ssh** command offer secure and robust remote connection terminals.

Tip

A Python- and Shell-like REPL (=READ + EVAL + PRINT + LOOP) environment for Perl is offered by the **reply** command from the **libreply-perl** (new) package and the **re.pl** command from the **libdevel-repl-perl** (old) package.

Tip

The **rlwrap** and **rlfe** commands add input line editing capability with history support to any interactive commands. E.g. "**rlwrap dash -i**" .

Chapter 8

More Examples

There is an old Latin saying: "**fabricando fit faber**" ("practice makes perfect").

It is highly recommended to practice and experiment with all the steps of Debian packaging with simple packages. This chapter provides you with many upstream cases for your practice.

This should also serve as introductory examples for many programing topics.

- Programing in the POSIX shell, Python3, and C.
- Method to create a desktop GUI program launcher with icon graphics.
- Conversion of a command from CLI to GUI.
- Conversion of a program to use **gettext** for internationalization and localization: POSIX shell, Python3, and C sources.
- Overview of many build systems: Makefile, Python distutils, Autotools, and CMake.

Please note that Debian takes a few things seriously:

- Free software (a.k.a. Libre software)
- Stability and security of OS
- Universal OS realized via:
  - free choice for upstream sources,
  - free choice of CPU architectures, and
  - free choice of UI languages.

The typical packaging example presented in Chapter 4 is the prerequisite for this chapter.

Some details are intentionally left vague in the following sections. Please try to read the pertinent documentation and practice yourself to find them out.

Tip

The best source of a packaging example is the current Debian archive itself. Please use the "Debian Code Search" service to find pertinent examples.

8.1 Cherry-pick templates

Here is an example of creating a simple Debian package from a zero content source on an empty directory.

This is a good platform to get all the template files without making a mess in the upstream source tree you are working on.

Let's assume this empty directory to be **debhello-0.1**.

```
 $ mkdir debhello-0.1
 $ tree
.
└── debhello-0.1

1 directory, 0 files
```

Let's generate the maximum amount of template files by specifying the **-x4** option.
Let's also use the "**-p** *debhello* **-t -u** *0.1* **-r** *1"* options to make the missing upstream tarball.

```
 $ debmake -t -p debhello -u 0.1 -r 1 -x4
I: set parameters
 ...
I: debmake -x "4" ...
I: creating => debian/control
I: creating => debian/copyright
I: substituting => /usr/share/debmake/extra0/rules
 ...
I: creating => debian/license-examples/GPL-2.0+
I: substituting => /usr/share/debmake/extra4/BSD-3-Clause
I: creating => debian/license-examples/BSD-3-Clause
I: substituting => /usr/share/debmake/extra4/Artistic-1.0
I: creating => debian/license-examples/Artistic-1.0
I: $ wrap-and-sort
```

Let's inspect generated template files.

```
 $ cd ..
 $ tree
.
├── debhello-0.1
│   └── debian
│       ├── README.Debian
│       ├── changelog
│       ├── clean
│       ├── compat
│       ├── control
│       ├── copyright
│       ├── debhello.bug-control.ex
│       ├── debhello.bug-presubj.ex
│       ├── debhello.bug-script.ex
│       ├── debhello.conffiles.ex
 ...
│       └── watch
├── debhello-0.1.tar.gz
└── debhello_0.1.orig.tar.gz -> debhello-0.1.tar.gz

5 directories, 51 files
```

Now you can copy any of these generated template files in the *debhello-0.1/***debian/** directory to your package as needed while renaming them as needed.

Tip

The generated template files can be made more verbose by invoking the **debmake** command with the **-T** option (tutorial mode).

8.2 No Makefile (shell, CLI)

Here is an example of creating a simple Debian package from a POSIX shell CLI program without its build system.

Let's assume this upstream tarball to be **debhello-0.2.tar.gz**.

This type of source has no automated means and files must be installed manually.

```
$ tar -xzmf debhello-0.2.tar.gz
$ cd debhello-0.2
$ sudo cp scripts/hello /bin/hello
...
```

Let's get the source and make the Debian package.

Download debhello-0.2.tar.gz

```
$ wget http://www.example.org/download/debhello-0.2.tar.gz
...
$ tar -xzmf debhello-0.2.tar.gz
$ tree
.
├── debhello-0.2
│   ├── LICENSE
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── man
│   │   └── hello.1
│   └── scripts
│       └── hello
└── debhello-0.2.tar.gz

4 directories, 6 files
```

Here, the POSIX shell script **hello** is a very simple one.

hello (v=0.2)

```
$ cat debhello-0.2/scripts/hello
#!/bin/sh -e
echo "Hello from the shell!"
echo ""
echo -n "Type Enter to exit this program: "
read X
```

Here, **hello.desktop** supports the Desktop Entry Specification.

hello.desktop (v=0.2)

```
$ cat debhello-0.2/data/hello.desktop
[Desktop Entry]
Name=Hello
Name[fr]=Bonjour
Comment=Greetings
Comment[fr]=Salutations
Type=Application
Keywords=hello
Exec=hello
Terminal=true
Icon=hello.png
Categories=Utility;
```

Here, **hello.png** is the icon graphics file.

Let's package this with the **debmake** command. Here, the **-b':sh'** option is used to specify that the generated binary package is a shell script.

```
 $ cd debhello-0.2
 $ debmake -b':sh'
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="0.2", rev="1"
I: *** start packaging in "debhello-0.2". ***
I: provide debhello_0.2.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-0.2.tar.gz debhello_0.2.orig.tar.gz
I: pwd = "/path/to/debhello-0.2"
I: parse binary package settings: :sh
I: binary package=debhello Type=script / Arch=all M-A=foreign
...
```

Let's inspect notable template files generated.

The source tree after the basic debmake execution. (v=0.2)

```
 $ cd ..
 $ tree
.
├── debhello-0.2
│   ├── LICENSE
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── debian
│   │   ├── README.Debian
│   │   ├── changelog
│   │   ├── compat
│   │   ├── control
│   │   ├── copyright
│   │   ├── patches
│   │   │   └── series
│   │   ├── rules
│   │   ├── source
│   │   │   ├── format
│   │   │   └── local-options
│   │   └── watch
│   ├── man
│   │   └── hello.1
│   └── scripts
│       └── hello
├── debhello-0.2.tar.gz
└── debhello_0.2.orig.tar.gz -> debhello-0.2.tar.gz

7 directories, 17 files
```

debian/rules (template file, v=0.2):

```
 $ cat debhello-0.2/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1

%:
        dh $@
```

This is essentially the standard **debian/rules** file with the **dh** command. Since this is the script package, this template **debian/rules** file has no build flag related contents.

debian/control (template file, v=0.2):

```
 $ cat debhello-0.2/debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: "Firstname Lastname" <email.address@example.org>
Build-Depends: debhelper (>=11~)
Standards-Version: 4.1.4
Homepage: <insert the upstream URL, if relevant>

Package: debhello
Architecture: all
Multi-Arch: foreign
Depends: ${misc:Depends}
Description: auto-generated package by debmake
 This Debian binary package was auto-generated by the
 debmake(1) command provided by the debmake package.
```

Since this is the shell script package, the **debmake** command sets "**Architecture: all**" and "**Multi-Arch: foreign**". Also, it sets required **substvar** parameters as "**Depends: ${misc:Depends}**". These are explained in Chapter 5.

Since this upstream source lacks the upstream **Makefile**, that functionality needs to be provided by the maintainer. This upstream source contains only a script file and data files and no C source files; the **build** process can be skipped but the **install** process needs to be implemented. For this case, this is achieved cleanly by adding the **debian/install** and **debian/manpages** files without complicating the **debian/rules** file.

Let's make this Debian package better as the maintainer.

debian/rules (maintainer version, v=0.2):

```
 $ vim debhello-0.2/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-0.2/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1

%:
        dh $@
```

debian/control (maintainer version, v=0.2):

```
 $ vim debhello-0.2/debian/control
 ... hack, hack, hack, ...
 $ cat debhello-0.2/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>=11~)
Standards-Version: 4.3.0
Homepage: https://salsa.debian.org/debian/debmake-doc

Package: debhello
Architecture: all
Multi-Arch: foreign
Depends: ${misc:Depends}
Description: example package in the debmake-doc package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
```

Warning

If you leave "**Section: unknown**" in the template **debian/control** file unchanged, the **lintian** error may cause a build failure.

debian/install (maintainer version, v=0.2):

```
 $ vim debhello-0.2/debian/install
 ... hack, hack, hack, ...
 $ cat debhello-0.2/debian/install
data/hello.desktop usr/share/applications
data/hello.png usr/share/pixmaps
scripts/hello usr/bin
```

debian/manpages (maintainer version, v=0.2):

```
 $ vim debhello-0.2/debian/manpages
 ... hack, hack, hack, ...
 $ cat debhello-0.2/debian/manpages
man/hello.1
```

There are several other template files under the **debian/** directory. These also need to be updated.
Template files under debian/. (v=0.2):

```
 $ tree debhello-0.2/debian
debhello-0.2/debian
├── README.Debian
├── changelog
├── compat
├── control
├── copyright
├── install
├── manpages
├── patches
│   └── series
├── rules
├── source
│   ├── format
│   └── local-options
└── watch

2 directories, 12 files
```

You can create a non-native Debian package using the **debuild** command (or its equivalents) in this source tree. The command output is very verbose and explains what it does as follows.

```
 $ cd debhello-0.2
 $ debuild
 dpkg-buildpackage -us -uc -ui -i
 ...
 fakeroot debian/rules clean
dh clean
 ...
 debian/rules build
dh build
   dh_update_autotools_config
   dh_autoreconf
```

```
   create-stamp debian/debhelper-build-stamp
 fakeroot debian/rules binary
dh binary
   dh_testroot
   dh_prep
        rm -f -- debian/debhello.substvars
        rm -fr -- debian/.debhelper/generated/debhello/ debian/debhello/ debi...
 ...
 fakeroot debian/rules binary
dh binary
 ...
```

Let's inspect the result.
The generated files of debhello version 0.2 by the debuild command:

```
 $ cd ..
 $ tree -FL 1
.
├── debhello-0.2/
├── debhello-0.2.tar.gz
├── debhello_0.2-1.debian.tar.xz
├── debhello_0.2-1.dsc
├── debhello_0.2-1_all.deb
├── debhello_0.2-1_amd64.build
├── debhello_0.2-1_amd64.buildinfo
├── debhello_0.2-1_amd64.changes
└── debhello_0.2.orig.tar.gz -> debhello-0.2.tar.gz

1 directory, 8 files
```

You see all the generated files.

- The **debhello_0.2.orig.tar.gz** file is a symlink to the upstream tarball.
- The **debhello_0.2-1.debian.tar.xz** file contains the maintainer generated contents.
- The **debhello_0.2-1.dsc** file is the meta data file for the Debian source package.
- The **debhello_0.2-1_all.deb** file is the Debian binary package.
- The **debhello_0.2-1_amd64.build** file is the build log file.
- The **debhello_0.2-1_amd64.buildinfo** file is the meta data file generated by **dpkg-genbuildinfo**(1).
- The **debhello_0.2-1_amd64.changes** file is the meta data file for the Debian binary package.

The **debhello_0.2-1.debian.tar.xz** file contains the Debian changes to the upstream source as follows.
The compressed archive contents of debhello_0.2-1.debian.tar.xz:

```
 $ tar -tzf debhello-0.2.tar.gz
debhello-0.2/
debhello-0.2/LICENSE
debhello-0.2/data/
debhello-0.2/data/hello.desktop
debhello-0.2/data/hello.png
debhello-0.2/scripts/
debhello-0.2/scripts/hello
debhello-0.2/man/
debhello-0.2/man/hello.1
 $ tar --xz -tf debhello_0.2-1.debian.tar.xz
debian/
debian/README.Debian
debian/changelog
debian/compat
```

```
debian/control
debian/copyright
debian/install
debian/manpages
debian/patches/
debian/patches/series
debian/rules
debian/source/
debian/source/format
debian/watch
```

The **debhello_0.2-1_amd64.deb** file contains the files to be installed as follows.

The binary package contents of debhello_0.2-1_all.deb:

```
 $ dpkg -c debhello_0.2-1_all.deb
drwxr-xr-x root/root ...  ./
drwxr-xr-x root/root ...  ./usr/
drwxr-xr-x root/root ...  ./usr/bin/
-rwxr-xr-x root/root ...  ./usr/bin/hello
drwxr-xr-x root/root ...  ./usr/share/
drwxr-xr-x root/root ...  ./usr/share/applications/
-rw-r--r-- root/root ...  ./usr/share/applications/hello.desktop
drwxr-xr-x root/root ...  ./usr/share/doc/
drwxr-xr-x root/root ...  ./usr/share/doc/debhello/
-rw-r--r-- root/root ...  ./usr/share/doc/debhello/README.Debian
-rw-r--r-- root/root ...  ./usr/share/doc/debhello/changelog.Debian.gz
-rw-r--r-- root/root ...  ./usr/share/doc/debhello/copyright
drwxr-xr-x root/root ...  ./usr/share/man/
drwxr-xr-x root/root ...  ./usr/share/man/man1/
-rw-r--r-- root/root ...  ./usr/share/man/man1/hello.1.gz
drwxr-xr-x root/root ...  ./usr/share/pixmaps/
-rw-r--r-- root/root ...  ./usr/share/pixmaps/hello.png
```

Here is the generated dependency list of **debhello_0.2-1_all.deb**.

The generated dependency list of debhello_0.2-1_all.deb:

```
 $ dpkg -f debhello_0.2-1_all.deb pre-depends depends recommends conflicts br...
```

8.3 Makefile (shell, CLI)

Here is an example of creating a simple Debian package from a POSIX shell CLI program using the **Makefile** as its build system.

Let's assume its upstream tarball to be **debhello-1.0.tar.gz**.

This type of source is meant to be installed as a non-system file as:

```
 $ tar -xzmf debhello-1.0.tar.gz
 $ cd debhello-1.0
 $ make install
```

Debian packaging requires changing this "**make install**" process to install files to the target system image location instead of the normal location under **/usr/local**.

Let's get the source and make the Debian package.

Download debhello-1.0.tar.gz

```
 $ wget http://www.example.org/download/debhello-1.0.tar.gz
 ...
 $ tar -xzmf debhello-1.0.tar.gz
 $ tree
.
```

```
├── debhello-1.0
│   ├── LICENSE
│   ├── Makefile
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── man
│   │   └── hello.1
│   └── scripts
│       └── hello
└── debhello-1.0.tar.gz

4 directories, 7 files
```

Here, the **Makefile** uses **$(DESTDIR)** and **$(prefix)** properly. All other files are the same as in Section 8.2 and most of the packaging activities are the same.

Makefile (v=1.0)

```
 $ cat debhello-1.0/Makefile
prefix = /usr/local

all:
        : # do nothing

install:
        install -D scripts/hello \
                $(DESTDIR)$(prefix)/bin/hello
        install -m 644 -D data/hello.desktop \
                $(DESTDIR)$(prefix)/share/applications/hello.desktop
        install -m 644 -D data/hello.png \
                $(DESTDIR)$(prefix)/share/pixmaps/hello.png
        install -m 644 -D man/hello.1 \
                $(DESTDIR)$(prefix)/share/man/man1/hello.1

clean:
        : # do nothing

distclean: clean

uninstall:
        -rm -f $(DESTDIR)$(prefix)/bin/hello
        -rm -f $(DESTDIR)$(prefix)/share/applications/hello.desktop
        -rm -f $(DESTDIR)$(prefix)/share/pixmaps/hello.png
        -rm -f $(DESTDIR)$(prefix)/share/man/man1/hello.1

.PHONY: all install clean distclean uninstall
```

Let's package this with the **debmake** command. Here, the **-b':sh'** option is used to specify that the generated binary package is a shell script.

```
 $ cd debhello-1.0
 $ debmake -b':sh'
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="1.0", rev="1"
I: *** start packaging in "debhello-1.0". ***
I: provide debhello_1.0.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.0.tar.gz debhello_1.0.orig.tar.gz
I: pwd = "/path/to/debhello-1.0"
I: parse binary package settings: :sh
I: binary package=debhello Type=script / Arch=all M-A=foreign
...
```

Let's inspect the notable template files generated.
debian/rules (template file, v=1.0):

```
 $ cat debhello-1.0/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1

%:
        dh $@

#override_dh_auto_install:
#       dh_auto_install -- prefix=/usr

#override_dh_install:
#       dh_install --list-missing -X.pyc -X.pyo
```

Let's make this Debian package better as the maintainer.
debian/rules (maintainer version, v=1.0):

```
 $ vim debhello-1.0/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-1.0/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1

%:
        dh $@

override_dh_auto_install:
        dh_auto_install -- prefix=/usr
```

Since this upstream source has the proper upstream **Makefile**, there is no need to create **debian/install** and **debian/manpages** files.

The **debian/control** file is exactly the same as the one in Section 8.2.

There are several other template files under the **debian/** directory. These also need to be updated.
Template files under debian/. (v=1.0):

```
 $ tree debhello-1.0/debian
debhello-1.0/debian
├── README.Debian
├── changelog
├── compat
├── control
├── copyright
├── patches
│   └── series
├── rules
├── source
│   ├── format
│   └── local-options
└── watch

2 directories, 10 files
```

The rest of the packaging activities are practically the same as the ones in Section 8.2.

8.4 setup.py (Python3, CLI)

Here is an example of creating a simple Debian package from a Python3 CLI program using **setup.py** as its build system.

Let's assume its upstream tarball to be **debhello-1.1.tar.gz**.
This type of source is meant to be installed as a non-system file as:

```
$ tar -xzmf debhello-1.1.tar.gz
$ cd debhello-1.1
$ python3 setup.py install
```

Debian packaging requires changing the last line to "**python3 setup.py install --install-layout=deb**" to install files into the target system image location. This issue is automatically addressed when using the **dh** command for Debian packaging.
Let's get the source and make the Debian package.
Download debhello-1.1.tar.gz

```
$ wget http://www.example.org/download/debhello-1.1.tar.gz
...
$ tar -xzmf debhello-1.1.tar.gz
$ tree
.
├── debhello-1.1
│   ├── LICENSE
│   ├── MANIFEST.in
│   ├── PKG-INFO
│   ├── hello_py
│   │   └── __init__.py
│   ├── scripts
│   │   └── hello
│   └── setup.py
└── debhello-1.1.tar.gz

3 directories, 7 files
```

Here, the **hello** script and its associated **hello_py** module are as follows.
hello (v=1.1)

```
$ cat debhello-1.1/scripts/hello
#!/usr/bin/python3
import hello_py

if __name__ == '__main__':
    hello_py.main()
```

hello_py/__init__.py (v=1.1)

```
$ cat debhello-1.1/hello_py/__init__.py
#!/usr/bin/python3
def main():
    print('Hello Python3!')
    input("Press Enter to continue...")
    return

if __name__ == '__main__':
    main()
```

These are packaged using the Python distutils with the **setup.py** and **MANIFEST.in** files.
setup.py (v=1.1)

```
$ cat debhello-1.1/setup.py
#!/usr/bin/python3
# vi:se ts=4 sts=4 et ai:
from distutils.core import setup
```

```
setup(name='debhello',
    version='4.0',
    description='Hello Python',
    long_description='Hello Python program.',
    author='Osamu Aoki',
    author_email='osamu@debian.org',
    url='http://people.debian.org/~osamu/',
    packages=['hello_py'],
    package_dir={'hello_py': 'hello_py'},
    scripts=['scripts/hello'],
    classifiers = ['Development Status :: 3 - Alpha',
        'Environment :: Console',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: MIT License',
        'Natural Language :: English',
        'Operating System :: POSIX :: Linux',
        'Programming Language :: Python :: 3',
        'Topic :: Utilities',
    ],
    platforms   = 'POSIX',
    license     = 'MIT License'
)
```

MANIFEST.in (v=1.1)

```
 $ cat debhello-1.1/MANIFEST.in
include MANIFEST.in
include LICENSE
```

Tip

Many modern Python packages are distributed using setuptools. Since setuptools is an enhanced alternative to distutils, this example is useful for them.

Let's package this with the **debmake** command. Here, the **-b':py3'** option is used to specify the generated binary package containing Python3 script and module files.

```
 $ cd debhello-1.1
 $ debmake -b':py3'
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="1.1", rev="1"
I: *** start packaging in "debhello-1.1". ***
I: provide debhello_1.1.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.1.tar.gz debhello_1.1.orig.tar.gz
I: pwd = "/path/to/debhello-1.1"
I: parse binary package settings: :py3
I: binary package=debhello Type=python3 / Arch=all M-A=foreign
...
```

Let's inspect the notable template files generated.

debian/rules (template file, v=1.1):

```
 $ cat debhello-1.1/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
```

```
#export DH_VERBOSE = 1

%:
        dh $@ --with python3 --buildsystem=pybuild
```

This is essentially the standard **debian/rules** file with the **dh** command.

The use of the "**--with python3**" option invokes **dh_python3** to calculate Python dependencies, add maintainer scripts to byte compiled files, etc. See **dh_python3**(1).

The use of the "**--buildsystem=pybuild**" option invokes various build systems for requested Python versions in order to build modules and extensions. See **pybuild**(1).

debian/control (template file, v=1.1):

```
 $ cat debhello-1.1/debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: "Firstname Lastname" <email.address@example.org>
Build-Depends: debhelper (>=11~), dh-python, python3-all
Standards-Version: 4.1.4
Homepage: <insert the upstream URL, if relevant>
X-Python3-Version: >= 3.2

Package: debhello
Architecture: all
Multi-Arch: foreign
Depends: ${misc:Depends}, ${python3:Depends}
Description: auto-generated package by debmake
 This Debian binary package was auto-generated by the
 debmake(1) command provided by the debmake package.
```

Since this is the Python3 package, the **debmake** command sets "**Architecture: all**" and "**Multi-Arch: foreign**". Also, it sets required **substvar** parameters as "**Depends: ${python3:Depends}, ${misc:Depends}**". These are explained in Chapter 5.

Let's make this Debian package better as the maintainer.

debian/rules (maintainer version, v=1.1):

```
 $ vim debhello-1.1/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-1.1/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1

%:
        dh $@ --with python3 --buildsystem=pybuild
```

debian/control (maintainer version, v=1.1):

```
 $ vim debhello-1.1/debian/control
 ... hack, hack, hack, ...
 $ cat debhello-1.1/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>= 12~), dh-python, python3-all
Standards-Version: 4.3.0
Homepage: https://salsa.debian.org/debian/debmake-doc
X-Python3-Version: >= 3.2

Package: debhello
Architecture: all
```

```
Multi-Arch: foreign
Depends: ${misc:Depends}, ${python3:Depends}
Description: example package in the debmake-doc package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
```

The **hello** command does not come with the upstream-provided manpage; let's add it as the maintainer.
debian/manpages etc. (maintainer version, v=1.1):

```
 $ vim debhello-1.1/debian/hello.1
 ... hack, hack, hack, ...
 $ vim debhello-1.1/debian/manpages
 ... hack, hack, hack, ...
 $ cat debhello-1.1/debian/manpages
debian/hello.1
```

There are several other template files under the **debian/** directory. These also need to be updated.
The rest of the packaging activities are practically the same as the ones in Section 8.3.
Template files under debian/. (v=1.1):

```
 $ tree debhello-1.1/debian
debhello-1.1/debian
├── README.Debian
├── changelog
├── compat
├── control
├── copyright
├── hello.1
├── manpages
├── patches
│   └── series
├── rules
├── source
│   ├── format
│   └── local-options
└── watch

2 directories, 12 files
```

Here is the generated dependency list of **debhello_1.1-1_all.deb**.
The generated dependency list of debhello_1.1-1_all.deb:

```
 $ dpkg -f debhello_1.1-1_all.deb pre-depends depends recommends conflicts br...
Depends: python3:any (>= 3.2~)
```

8.5 Makefile (shell, GUI)

Here is an example of creating a simple Debian package from a POSIX shell GUI program using the **Makefile** as its build system.

This upstream is based on Section 8.3 with enhanced GUI support.
Let's assume its upstream tarball to be **debhello-1.2.tar.gz**.
Let's get the source and make the Debian package.
Download debhello-1.2.tar.gz

```
 $ wget http://www.example.org/download/debhello-1.2.tar.gz
 ...
 $ tar -xzmf debhello-1.2.tar.gz
 $ tree
.
├── debhello-1.2
│   ├── LICENSE
│   ├── Makefile
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── man
│   │   └── hello.1
│   └── scripts
│       └── hello
└── debhello-1.2.tar.gz

4 directories, 7 files
```

Here, the **hello** has been re-written to use the **zenity** command to make this a GTK+ GUI program.
hello (v=1.2)

```
 $ cat debhello-1.2/scripts/hello
#!/bin/sh -e
zenity --info --title "hello" --text "Hello from the shell!"
```

Here, the desktop file is updated to be **Terminal=false** as a GUI program.
hello.desktop (v=1.2)

```
 $ cat debhello-1.2/data/hello.desktop
[Desktop Entry]
Name=Hello
Name[fr]=Bonjour
Comment=Greetings
Comment[fr]=Salutations
Type=Application
Keywords=hello
Exec=hello
Terminal=false
Icon=hello.png
Categories=Utility;
```

All other files are the same as in Section 8.3.

Let's package this with the **debmake** command. Here, the **-b':sh'** option is used to specify that the generated binary package is a shell script.

```
 $ cd debhello-1.2
 $ debmake -b':sh'
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="1.2", rev="1"
I: *** start packaging in "debhello-1.2". ***
I: provide debhello_1.2.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.2.tar.gz debhello_1.2.orig.tar.gz
I: pwd = "/path/to/debhello-1.2"
I: parse binary package settings: :sh
I: binary package=debhello Type=script / Arch=all M-A=foreign
...
```

Let's inspect the notable template files generated.

debian/control (template file, v=1.2):

```
 $ cat debhello-1.2/debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: "Firstname Lastname" <email.address@example.org>
Build-Depends: debhelper (>=11~)
Standards-Version: 4.1.4
Homepage: <insert the upstream URL, if relevant>

Package: debhello
Architecture: all
Multi-Arch: foreign
Depends: ${misc:Depends}
Description: auto-generated package by debmake
 This Debian binary package was auto-generated by the
 debmake(1) command provided by the debmake package.
```

Let's make this Debian package better as the maintainer.
debian/control (maintainer version, v=1.2):

```
 $ vim debhello-1.2/debian/control
 ... hack, hack, hack, ...
 $ cat debhello-1.2/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>=11~)
Standards-Version: 4.3.0
Homepage: https://salsa.debian.org/debian/debmake-doc

Package: debhello
Architecture: all
Multi-Arch: foreign
Depends: zenity, ${misc:Depends}
Description: example package in the debmake-doc package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
```

Please note the manually added **zenity** dependency.
The **debian/rules** file is exactly the same as the one in Section 8.3.
There are several other template files under the **debian/** directory. These also need to be updated.
Template files under debian/. (v=1.2):

```
 $ tree debhello-1.2/debian
debhello-1.2/debian
├── README.Debian
├── changelog
├── compat
├── control
├── copyright
├── patches
│   └── series
├── rules
├── source
│   ├── format
│   └── local-options
```

```
└── watch

2 directories, 10 files
```

The rest of the packaging activities are practically the same as in Section 8.3.
Here is the generated dependency list of **debhello_1.2-1_all.deb**.
The generated dependency list of debhello_1.2-1_all.deb:

```
 $ dpkg -f debhello_1.2-1_all.deb pre-depends depends recommends conflicts br...
Depends: zenity
```

8.6 setup.py (Python3, GUI)

Here is an example of creating a simple Debian package from a Python3 GUI program using the **setup.py** as its build system.
This upstream is based on Section 8.4 with enhanced GUI, desktop icon, and manpage support.
Let's assume this upstream tarball to be **debhello-1.3.tar.gz**.
Let's get the source and make the Debian package.
Download debhello-1.3.tar.gz

```
 $ wget http://www.example.org/download/debhello-1.3.tar.gz
 ...
 $ tar -xzmf debhello-1.3.tar.gz
 $ tree
.
├── debhello-1.3
│   ├── LICENSE
│   ├── MANIFEST.in
│   ├── PKG-INFO
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── hello_py
│   │   └── __init__.py
│   ├── man
│   │   └── hello.1
│   ├── scripts
│   │   └── hello
│   └── setup.py
└── debhello-1.3.tar.gz

5 directories, 10 files
```

Here are the upstream sources.
hello (v=1.3)

```
 $ cat debhello-1.3/scripts/hello
#!/usr/bin/python3
import hello_py

if __name__ == '__main__':
    hello_py.main()
```

hello_py/__init__.py (v=1.3)

```
 $ cat debhello-1.3/hello_py/__init__.py
#!/usr/bin/python3
from gi.repository import Gtk
```

```
class TopWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self)
        self.title = "Hello World!"
        self.counter = 0
        self.border_width = 10
        self.set_default_size(400, 100)
        self.set_position(Gtk.WindowPosition.CENTER)
        self.button = Gtk.Button(label="Click me!")
        self.button.connect("clicked", self.on_button_clicked)
        self.add(self.button)
        self.connect("delete-event", self.on_window_destroy)

    def on_window_destroy(self, *args):
        Gtk.main_quit(*args)

    def on_button_clicked(self, widget):
        self.counter += 1
        widget.set_label("Hello, World!\nClick count = %i" % self.counter)

def main():
    window = TopWindow()
    window.show_all()
    Gtk.main()

if __name__ == '__main__':
    main()
```

setup.py (v=1.3)

```
 $ cat debhello-1.3/setup.py
#!/usr/bin/python3
# vi:se ts=4 sts=4 et ai:
from distutils.core import setup

setup(name='debhello',
    version='4.1',
    description='Hello Python',
    long_description='Hello Python program.',
    author='Osamu Aoki',
    author_email='osamu@debian.org',
    url='http://people.debian.org/~osamu/',
    packages=['hello_py'],
    package_dir={'hello_py': 'hello_py'},
    scripts=['scripts/hello'],
    data_files=[
        ('share/applications', ['data/hello.desktop']),
        ('share/pixmaps', ['data/hello.png']),
        ('share/man/man1', ['man/hello.1']),
    ],
    classifiers = ['Development Status :: 3 - Alpha',
        'Environment :: Console',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: MIT License',
        'Natural Language :: English',
        'Operating System :: POSIX :: Linux',
        'Programming Language :: Python :: 3',
        'Topic :: Utilities',
    ],
    platforms   = 'POSIX',
    license     = 'MIT License'
)
```

MANIFEST.in (v=1.3)

```
 $ cat debhello-1.3/MANIFEST.in
include MANIFEST.in
include LICENSE
include data/hello.deskto
include data/hello.png
include man/hello.1
```

Let's package this with the **debmake** command. Here, the **-b':py3'** option is used to specify that the generated binary package contains Python3 script and module files.

```
 $ cd debhello-1.3
 $ debmake -b':py3'
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="1.3", rev="1"
I: *** start packaging in "debhello-1.3". ***
I: provide debhello_1.3.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.3.tar.gz debhello_1.3.orig.tar.gz
I: pwd = "/path/to/debhello-1.3"
I: parse binary package settings: :py3
I: binary package=debhello Type=python3 / Arch=all M-A=foreign
...
```

The result is practically the same as in Section 8.4.

Let's make this Debian package better as the maintainer.

debian/rules (maintainer version, v=1.3):

```
 $ vim debhello-1.3/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-1.3/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1

%:
        dh $@ --with python3 --buildsystem=pybuild
```

debian/control (maintainer version, v=1.3):

```
 $ vim debhello-1.3/debian/control
 ... hack, hack, hack, ...
 $ cat debhello-1.3/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>=11~), dh-python, python3-all
Standards-Version: 4.3.0
Homepage: https://salsa.debian.org/debian/debmake-doc
X-Python3-Version: >= 3.2

Package: debhello
Architecture: all
Multi-Arch: foreign
Depends: gir1.2-gtk-3.0, python3-gi, ${misc:Depends}, ${python3:Depends}
Description: example package in the debmake-doc package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
```

```
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
```

Please note the manually added **python3-gi** and **gir1.2-gtk-3.0** dependencies.

Since this upstream source has a manpage and other files with matching entries in the **setup.py** file, there is no need to create them and add the **debian/install** and **debian/manpages** files that were required in Section 8.4.

The rest of the packaging activities are practically the same as in Section 8.4.

Here is the generated dependency list of **debhello_1.3-1_all.deb**.

The generated dependency list of debhello_1.3-1_all.deb:

```
 $ dpkg -f debhello_1.3-1_all.deb pre-depends depends recommends conflicts br...
Depends: gir1.2-gtk-3.0, python3-gi, python3:any (>= 3.2~)
```

8.7 Makefile (single-binary package)

Here is an example of creating a simple Debian package from a simple C source program using the **Makefile** as its build system.

This is an enhanced upstream source example for Chapter 4. This comes with the manpage, the desktop file, and the desktop icon. This also links to an external library **libm** to be a more practical example.

Let's assume this upstream tarball to be **debhello-1.4.tar.gz**.

This type of source is meant to be installed as a non-system file as:

```
 $ tar -xzmf debhello-1.4.tar.gz
 $ cd debhello-1.4
 $ make
 $ make install
```

Debian packaging requires changing this "**make install**" process to install files into the target system image location instead of the normal location under **/usr/local**.

Let's get the source and make the Debian package.

Download debhello-1.4.tar.gz

```
 $ wget http://www.example.org/download/debhello-1.4.tar.gz
 ...
 $ tar -xzmf debhello-1.4.tar.gz
 $ tree
.
├── debhello-1.4
│   ├── LICENSE
│   ├── Makefile
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── man
│   │   └── hello.1
│   └── src
│       ├── config.h
│       └── hello.c
└── debhello-1.4.tar.gz

4 directories, 8 files
```

Here, the contents of this source are as follows.

src/hello.c (v=1.4):

```
 $ cat debhello-1.4/src/hello.c
#include "config.h"
#include <math.h>
```

```
#include <stdio.h>
int
main()
{
        printf("Hello, I am " PACKAGE_AUTHOR "!\n");
        printf("4.0 * atan(1.0) = %10f8\n", 4.0*atan(1.0));
        return 0;
}
```

src/config.h (v=1.4):

```
 $ cat debhello-1.4/src/config.h
#define PACKAGE_AUTHOR "Osamu Aoki"
```

Makefile (v=1.4):

```
 $ cat debhello-1.4/Makefile
prefix = /usr/local

all: src/hello

src/hello: src/hello.c
        $(CC) $(CPPFLAGS) $(CFLAGS) $(LDFLAGS) -o $@ $^ -lm

install: src/hello
        install -D src/hello \
                $(DESTDIR)$(prefix)/bin/hello
        install -m 644 -D data/hello.desktop \
                $(DESTDIR)$(prefix)/share/applications/hello.desktop
        install -m 644 -D data/hello.png \
                $(DESTDIR)$(prefix)/share/pixmaps/hello.png
        install -m 644 -D man/hello.1 \
                $(DESTDIR)$(prefix)/share/man/man1/hello.1

clean:
        -rm -f src/hello

distclean: clean

uninstall:
        -rm -f $(DESTDIR)$(prefix)/bin/hello
        -rm -f $(DESTDIR)$(prefix)/share/applications/hello.desktop
        -rm -f $(DESTDIR)$(prefix)/share/pixmaps/hello.png
        -rm -f $(DESTDIR)$(prefix)/share/man/man1/hello.1

.PHONY: all install clean distclean uninstall
```

Please note that this **Makefile** has the proper **install** target for the manpage, the desktop file, and the desktop icon.

Let's package this with the **debmake** command.

```
 $ cd debhello-1.4
 $ debmake
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="1.4", rev="1"
I: *** start packaging in "debhello-1.4". ***
I: provide debhello_1.4.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.4.tar.gz debhello_1.4.orig.tar.gz
I: pwd = "/path/to/debhello-1.4"
I: parse binary package settings:
```

```
I: binary package=debhello Type=bin / Arch=any M-A=foreign
...
```

The result is practically the same as in Section 4.5.

Let's make this Debian package, which is practically the same as in Section 4.6, better as the maintainer.

If the **DEB_BUILD_MAINT_OPTIONS** environment variable is not exported in **debian/rules**, lintian warns "W: debhello: hardening-no-relro usr/bin/hello" for the linking of **libm**.

The **debian/control** file makes it exactly the same as the one in Section 4.6, since the **libm** library is always available as a part of **libc6** (Priority: required).

There are several other template files under the **debian/** directory. These also need to be updated.

Template files under debian/. (v=1.4):

```
 $ tree debhello-1.4/debian
debhello-1.4/debian
├── README.Debian
├── changelog
├── compat
├── control
├── copyright
├── patches
│   └── series
├── rules
├── source
│   ├── format
│   └── local-options
└── watch

2 directories, 10 files
```

The rest of the packaging activities are practically the same as the one in Section 4.7.

Here is the generated dependency list of all binary packages.

The generated dependency list of all binary packages (v=1.4):

```
 $ dpkg -f debhello-dbgsym_1.4-1_amd64.deb pre-depends depends recommends con...
Depends: debhello (= 1.4-1)
 $ dpkg -f debhello_1.4-1_amd64.deb pre-depends depends recommends conflicts ...
Depends: libc6 (>= 2.3.4)
```

8.8 Makefile.in + configure (single-binary package)

Here is an example of creating a simple Debian package from a simple C source program using **Makefile.in** and **configure** as its build system.

This is an enhanced upstream source example for Section 8.7. This also links to an external library, **libm**, and this source is configurable using arguments to the **configure** script, which generates the **Makefile** and **src/config.h** files.

Let's assume this upstream tarball to be **debhello-1.5.tar.gz**.

This type of source is meant to be installed as a non-system file, for example, as:

```
 $ tar -xzmf debhello-1.5.tar.gz
 $ cd debhello-1.5
 $ ./configure --with-math
 $ make
 $ make install
```

Let's get the source and make the Debian package.

Download debhello-1.5.tar.gz

```
 $ wget http://www.example.org/download/debhello-1.5.tar.gz
 ...
 $ tar -xzmf debhello-1.5.tar.gz
 $ tree
.
├── debhello-1.5
│   ├── LICENSE
│   ├── Makefile.in
│   ├── configure
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── man
│   │   └── hello.1
│   └── src
│       └── hello.c
└── debhello-1.5.tar.gz

4 directories, 8 files
```

Here, the contents of this source are as follows.

src/hello.c (v=1.5):

```
 $ cat debhello-1.5/src/hello.c
#include "config.h"
#ifdef WITH_MATH
#  include <math.h>
#endif
#include <stdio.h>
int
main()
{
        printf("Hello, I am " PACKAGE_AUTHOR "!\n");
#ifdef WITH_MATH
        printf("4.0 * atan(1.0) = %10f8\n", 4.0*atan(1.0));
#else
        printf("I can't do MATH!\n");
#endif
        return 0;
}
```

Makefile.in (v=1.5):

```
 $ cat debhello-1.5/Makefile.in
prefix = @prefix@

all: src/hello

src/hello: src/hello.c
        $(CC) @VERBOSE@ \
                $(CPPFLAGS) \
                $(CFLAGS) \
                $(LDFLAGS) \
                -o $@ $^ \
                @LINKLIB@

install: src/hello
        install -D src/hello \
                $(DESTDIR)$(prefix)/bin/hello
        install -m 644 -D data/hello.desktop \
                $(DESTDIR)$(prefix)/share/applications/hello.desktop
        install -m 644 -D data/hello.png \
```

```
                $(DESTDIR)$(prefix)/share/pixmaps/hello.png
        install -m 644 -D man/hello.1 \
                $(DESTDIR)$(prefix)/share/man/man1/hello.1

clean:
        -rm -f src/hello

distclean: clean

uninstall:
        -rm -f $(DESTDIR)$(prefix)/bin/hello
        -rm -f $(DESTDIR)$(prefix)/share/applications/hello.desktop
        -rm -f $(DESTDIR)$(prefix)/share/pixmaps/hello.png
        -rm -f $(DESTDIR)$(prefix)/share/man/man1/hello.1

.PHONY: all install clean distclean uninstall
```

configure (v=1.5):

```
 $ cat debhello-1.5/configure
#!/bin/sh -e
# default values
PREFIX="/usr/local"
VERBOSE=""
WITH_MATH="0"
LINKLIB=""
PACKAGE_AUTHOR="John Doe"

# parse arguments
while [ "${1}" != "" ]; do
  VAR="${1%=*}" # Drop suffix =*
  VAL="${1#*=}" # Drop prefix *=
  case "${VAR}" in
  --prefix)
    PREFIX="${VAL}"
    ;;
  --verbose|-v)
    VERBOSE="-v"
    ;;
  --with-math)
    WITH_MATH="1"
    LINKLIB="-lm"
    ;;
  --author)
    PACKAGE_AUTHOR="${VAL}"
    ;;
  *)
    echo "W: Unknown argument: ${1}"
  esac
  shift
done

# setup configured Makefile and src/config.h
sed -e "s,@prefix@,${PREFIX}," \
    -e "s,@VERBOSE@,${VERBOSE}," \
    -e "s,@LINKLIB@,${LINKLIB}," \
    <Makefile.in >Makefile
if [ "${WITH_MATH}" = 1 ]; then
echo "#define WITH_MATH" >src/config.h
else
echo "/* not defined: WITH_MATH */" >src/config.h
fi
echo "#define PACKAGE_AUTHOR \"${PACKAGE_AUTHOR}\"" >>src/config.h
```

Please note that the **configure** command replaces strings with **@…@** in **Makefile.in** to produce **Makefile** and creates **src/config.h**.

Let's package this with the **debmake** command.

```
 $ cd debhello-1.5
 $ debmake
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="1.5", rev="1"
I: *** start packaging in "debhello-1.5". ***
I: provide debhello_1.5.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.5.tar.gz debhello_1.5.orig.tar.gz
I: pwd = "/path/to/debhello-1.5"
I: parse binary package settings:
I: binary package=debhello Type=bin / Arch=any M-A=foreign
...
```

The result is similar to Section 4.5 but not exactly the same.

Let's inspect the notable template files generated.

debian/rules (template file, v=1.5):

```
 $ cat debhello-1.5/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@
```

Let's make this Debian package better as the maintainer.

debian/rules (maintainer version, v=1.5):

```
 $ vim debhello-1.5/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-1.5/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@

override_dh_auto_configure:
        dh_auto_configure -- \
                --with-math \
                --author="Osamu Aoki"
```

There are several other template files under the **debian/** directory. These also need to be updated.

The rest of the packaging activities are practically the same as the one in Section 4.7.

8.9 Autotools (single-binary package)

Here is an example of creating a simple Debian package from a simple C source program using Autotools = Autoconf and Automake (**Makefile.am** and **configure.ac**) as its build system. See Section 5.16.1.

This source usually comes with the upstream auto-generated **Makefile.in** and **configure** files, too. This source can be packaged using these files as in Section 8.8 with the help of the **autotools-dev** package.

The better alternative is to regenerate these files using the latest Autoconf and Automake packages if the upstream provided **Makefile.am** and **configure.ac** are compatible with the latest version. This is advantageous for porting to new CPU architectures, etc. This can be automated by using the "**--with autoreconf**" option for the **dh** command.

Let's assume this upstream tarball to be **debhello-1.6.tar.gz**.

This type of source is meant to be installed as a non-system file, for example, as:

```
$ tar -xzmf debhello-1.6.tar.gz
$ cd debhello-1.6
$ autoreconf -ivf # optional
$ ./configure --with-math
$ make
$ make install
```

Let's get the source and make the Debian package.

Download debhello-1.6.tar.gz

```
$ wget http://www.example.org/download/debhello-1.6.tar.gz
...
$ tar -xzmf debhello-1.6.tar.gz
$ tree
.
├── debhello-1.6
│   ├── Makefile.am
│   ├── configure.ac
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── man
│   │   ├── Makefile.am
│   │   └── hello.1
│   └── src
│       ├── Makefile.am
│       └── hello.c
└── debhello-1.6.tar.gz

4 directories, 9 files
```

Here, the contents of this source are as follows.

src/hello.c (v=1.6):

```
$ cat debhello-1.6/src/hello.c
#include "config.h"
#ifdef WITH_MATH
#  include <math.h>
#endif
#include <stdio.h>
int
main()
{
        printf("Hello, I am " PACKAGE_AUTHOR "!\n");
#ifdef WITH_MATH
        printf("4.0 * atan(1.0) = %10f8\n", 4.0*atan(1.0));
#else
        printf("I can't do MATH!\n");
#endif
        return 0;
}
```

Makefile.am (v=1.6):

```
 $ cat debhello-1.6/Makefile.am
SUBDIRS = src man
 $ cat debhello-1.6/man/Makefile.am
dist_man_MANS = hello.1
 $ cat debhello-1.6/src/Makefile.am
bin_PROGRAMS = hello
hello_SOURCES = hello.c
```

configure.ac (v=1.6):

```
 $ cat debhello-1.6/configure.ac
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ([2.69])
AC_INIT([debhello],[2.1],[foo@example.org])
AC_CONFIG_SRCDIR([src/hello.c])
AC_CONFIG_HEADERS([config.h])
echo "Standard customization chores"
AC_CONFIG_AUX_DIR([build-aux])
AM_INIT_AUTOMAKE([foreign])
# Add #define PACKAGE_AUTHOR ... in config.h with a comment
AC_DEFINE(PACKAGE_AUTHOR, ["Osamu Aoki"], [Define PACKAGE_AUTHOR])
echo "Add --with-math option functionality to ./configure"
AC_ARG_WITH([math],
  [AS_HELP_STRING([--with-math],
    [compile with math library  @<:@default=yes@:>@])],
  [],
  [with_math="yes"]
  )
echo "==== withval   := \"$withval\""
echo "==== with_math := \"$with_math\""
# m4sh if-else construct
AS_IF([test "x$with_math" != "xno"],[
  echo "==== Check include: math.h"
  AC_CHECK_HEADER(math.h,[],[
    AC_MSG_ERROR([Couldn't find math.h.] )
  ])
  echo "==== Check library: libm"
  AC_SEARCH_LIBS(atan, [m])
  #AC_CHECK_LIB(m, atan)
  echo "==== Build with LIBS := \"$LIBS\""
  AC_DEFINE(WITH_MATH, [1], [Build with the math library])
],[
  echo "==== Skip building with math.h."
  AH_TEMPLATE(WITH_MATH, [Build without the math library])
])
# Checks for programs.
AC_PROG_CC
AC_CONFIG_FILES([Makefile
                 man/Makefile
                 src/Makefile])
AC_OUTPUT
```

Tip

Without "**foreign**" strictness level specified in **AM_INIT_AUTOMAKE()** as above, **automake** defaults to "**gnu**" strictness level requiring several files in the top-level directory. See "3.2 Strictness" in the **automake** document.

Let's package this with the **debmake** command.

```
 $ cd debhello-1.6
 $ debmake
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="1.6", rev="1"
I: *** start packaging in "debhello-1.6". ***
I: provide debhello_1.6.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.6.tar.gz debhello_1.6.orig.tar.gz
I: pwd = "/path/to/debhello-1.6"
I: parse binary package settings:
I: binary package=debhello Type=bin / Arch=any M-A=foreign
...
```

The result is similar to Section 8.8 but not exactly the same.
Let's inspect the notable template files generated.
debian/rules (template file, v=1.6):

```
 $ cat debhello-1.6/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@ --with autoreconf

#override_dh_install:
#       dh_install --list-missing -X.la -X.pyc -X.pyo
```

Let's make this Debian package better as the maintainer.
debian/rules (maintainer version, v=1.6):

```
 $ vim debhello-1.6/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-1.6/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@ --with autoreconf

override_dh_auto_configure:
        dh_auto_configure -- \
                --with-math
```

There are several other template files under the **debian/** directory. These also need to be updated.
The rest of the packaging activities are practically the same as the one in Section 4.7.

8.10 CMake (single-binary package)

Here is an example of creating a simple Debian package from a simple C source program using CMake (**CMakeLists.txt** and some files such as **config.h.in**) as its build system. See Section 5.16.2.

The **cmake** command generates the **Makefile** file based on the **CMakeLists.txt** file and its **-D** option. It also configures the file as specified in its **configure_file(…)** by replacing strings with **@…@** and changing the **#cmakedefine …** line.

Let's assume this upstream tarball to be **debhello-1.7.tar.gz**.

This type of source is meant to be installed as a non-system file, for example, as:

```
$ tar -xzmf debhello-1.7.tar.gz
$ cd debhello-1.7
$ mkdir obj-x86_64-linux-gnu # for out-of-tree build
$ cd obj-x86_64-linux-gnu
$ cmake ..
$ make
$ make install
```

Let's get the source and make the Debian package.

Download debhello-1.7.tar.gz

```
$ wget http://www.example.org/download/debhello-1.7.tar.gz
...
$ tar -xzmf debhello-1.7.tar.gz
$ tree
.
├── debhello-1.7
│   ├── CMakeLists.txt
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── man
│   │   ├── CMakeLists.txt
│   │   └── hello.1
│   └── src
│       ├── CMakeLists.txt
│       ├── config.h.in
│       └── hello.c
└── debhello-1.7.tar.gz

4 directories, 9 files
```

Here, the contents of this source are as follows.

src/hello.c (v=1.7):

```
$ cat debhello-1.7/src/hello.c
#include "config.h"
#ifdef WITH_MATH
#  include <math.h>
#endif
#include <stdio.h>
int
main()
{
        printf("Hello, I am " PACKAGE_AUTHOR "!\n");
#ifdef WITH_MATH
        printf("4.0 * atan(1.0) = %10f8\n", 4.0*atan(1.0));
#else
        printf("I can't do MATH!\n");
#endif
        return 0;
}
```

src/config.h.in (v=1.7):

```
 $ cat debhello-1.7/src/config.h.in
/* name of the package author */
#define PACKAGE_AUTHOR "@PACKAGE_AUTHOR@"
/* math library support */
#cmakedefine WITH_MATH
```

CMakeLists.txt (v=1.7):

```
 $ cat debhello-1.7/CMakeLists.txt
cmake_minimum_required(VERSION 2.8)
project(debhello)
set(PACKAGE_AUTHOR "Osamu Aoki")
add_subdirectory(src)
add_subdirectory(man)
 $ cat debhello-1.7/man/CMakeLists.txt
install(
  FILES ${CMAKE_CURRENT_SOURCE_DIR}/hello.1
  DESTINATION share/man/man1
)
 $ cat debhello-1.7/src/CMakeLists.txt
# Always define HAVE_CONFIG_H
add_definitions(-DHAVE_CONFIG_H)
# Interactively define WITH_MATH
option(WITH_MATH "Build with math support" OFF)
#variable_watch(WITH_MATH)
# Generate config.h from config.h.in
configure_file(
  "${CMAKE_CURRENT_SOURCE_DIR}/config.h.in"
  "${CMAKE_CURRENT_BINARY_DIR}/config.h"
)
include_directories("${CMAKE_CURRENT_BINARY_DIR}")
add_executable(hello hello.c)
install(TARGETS hello
  RUNTIME DESTINATION bin
)
```

Let's package this with the **debmake** command.

```
 $ cd debhello-1.7
 $ debmake
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="1.7", rev="1"
I: *** start packaging in "debhello-1.7". ***
I: provide debhello_1.7.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.7.tar.gz debhello_1.7.orig.tar.gz
I: pwd = "/path/to/debhello-1.7"
I: parse binary package settings:
I: binary package=debhello Type=bin / Arch=any M-A=foreign
...
```

The result is similar to Section 8.8 but not exactly the same.

Let's inspect the notable template files generated.

debian/rules (template file, v=1.7):

```
 $ cat debhello-1.7/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
```

```
#export DH_VERBOSE = 1
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@

#override_dh_auto_configure:
#       dh_auto_configure -- \
#             -DCMAKE_LIBRARY_ARCHITECTURE="$(DEB_TARGET_MULTIARCH)"
```

debian/control (template file, v=1.7):

```
 $ cat debhello-1.7/debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: "Firstname Lastname" <email.address@example.org>
Build-Depends: cmake, debhelper (>=11~)
Standards-Version: 4.1.4
Homepage: <insert the upstream URL, if relevant>

Package: debhello
Architecture: any
Multi-Arch: foreign
Depends: ${misc:Depends}, ${shlibs:Depends}
Description: auto-generated package by debmake
 This Debian binary package was auto-generated by the
 debmake(1) command provided by the debmake package.
```

Let's make this Debian package better as the maintainer.
debian/rules (maintainer version, v=1.7):

```
 $ vim debhello-1.7/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-1.7/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@

override_dh_auto_configure:
        dh_auto_configure -- -DWITH-MATH=1
```

debian/control (maintainer version, v=1.7):

```
 $ vim debhello-1.7/debian/control
 ... hack, hack, hack, ...
 $ cat debhello-1.7/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: cmake, debhelper (>=11~)
Standards-Version: 4.3.0
Homepage: https://salsa.debian.org/debian/debmake-doc
```

```
Package: debhello
Architecture: any
Multi-Arch: foreign
Depends: ${misc:Depends}, ${shlibs:Depends}
Description: example package in the debmake-doc package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
```

There are several other template files under the **debian/** directory. These also need to be updated.
The rest of the packaging activities are practically the same as the one in Section 8.8.

8.11 Autotools (multi-binary package)

Here is an example of creating a set of Debian binary packages including the executable package, the shared library package, the development file package, and the debug symbol package from a simple C source program using Autotools = Autoconf and Automake (which use **Makefile.am** and **configure.ac** as their input files) as its build system. See Section 5.16.1.

Let's package this in the same way as in Section 8.9.
Let's assume this upstream tarball to be **debhello-2.0.tar.gz**.
This type of source is meant to be installed as a non-system file, for example, as:

```
$ tar -xzmf debhello-2.0.tar.gz
$ cd debhello-2.0
$ autoreconf -ivf # optional
$ ./configure --with-math
$ make
$ make install
```

Let's get the source and make the Debian package.
Download debhello-2.0.tar.gz

```
$ wget http://www.example.org/download/debhello-2.0.tar.gz
...
$ tar -xzmf debhello-2.0.tar.gz
$ tree
.
├── debhello-2.0
│   ├── Makefile.am
│   ├── configure.ac
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── lib
│   │   ├── Makefile.am
│   │   ├── sharedlib.c
│   │   └── sharedlib.h
│   ├── man
│   │   ├── Makefile.am
│   │   └── hello.1
│   └── src
│       ├── Makefile.am
│       └── hello.c
└── debhello-2.0.tar.gz

5 directories, 12 files
```

Here, the contents of this source are as follows.
src/hello.c (v=2.0):

```
 $ cat debhello-2.0/src/hello.c
#include "config.h"
#include <stdio.h>
#include <sharedlib.h>
int
main()
{
        printf("Hello, I am " PACKAGE_AUTHOR "!\n");
        sharedlib();
        return 0;
}
```

lib/sharedlib.h and lib/sharedlib.c (v=1.6):

```
 $ cat debhello-2.0/lib/sharedlib.h
int sharedlib();
 $ cat debhello-2.0/lib/sharedlib.c
#include <stdio.h>
int
sharedlib()
{
        printf("This is a shared library!\n");
        return 0;
}
```

Makefile.am (v=2.0):

```
 $ cat debhello-2.0/Makefile.am
# recursively process `Makefile.am` in SUBDIRS
SUBDIRS = lib src man
 $ cat debhello-2.0/man/Makefile.am
# manpages (distributed in the source package)
dist_man_MANS = hello.1
 $ cat debhello-2.0/lib/Makefile.am
# libtool librares to be produced
lib_LTLIBRARIES = libsharedlib.la

# source files used for lib_LTLIBRARIES
libsharedlib_la_SOURCES = sharedlib.c

# C pre-processor flags used for lib_LTLIBRARIES
#libsharedlib_la_CPPFLAGS =

# Headers files to be installed in <prefix>/include
include_HEADERS = sharedlib.h

# Versioning Libtool Libraries with version triplets
libsharedlib_la_LDFLAGS = -version-info 1:0:0
 $ cat debhello-2.0/src/Makefile.am
# program executables to be produced
bin_PROGRAMS = hello

# source files used for bin_PROGRAMS
hello_SOURCES = hello.c

# C pre-processor flags used for bin_PROGRAMS
AM_CPPFLAGS = -I$(srcdir) -I$(top_srcdir)/lib

# Extra options for the linker for hello
# hello_LDFLAGS =
```

```
# Libraries the `hello` binary to be linked
hello_LDADD = $(top_srcdir)/lib/libsharedlib.la
```

configure.ac (v=2.0):

```
 $ cat debhello-2.0/configure.ac
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ([2.69])
AC_INIT([debhello],[2.2],[foo@example.org])
AC_CONFIG_SRCDIR([src/hello.c])
AC_CONFIG_HEADERS([config.h])
echo "Standard customization chores"
AC_CONFIG_AUX_DIR([build-aux])

AM_INIT_AUTOMAKE([foreign])

# Set default to --enable-shared --disable-static
LT_INIT([shared disable-static])

# find the libltdl sources in the libltdl sub-directory
LT_CONFIG_LTDL_DIR([libltdl])

# choose one
LTDL_INIT([recursive])
#LTDL_INIT([subproject])
#LTDL_INIT([nonrecursive])

# Add #define PACKAGE_AUTHOR ... in config.h with a comment
AC_DEFINE(PACKAGE_AUTHOR, ["Osamu Aoki"], [Define PACKAGE_AUTHOR])
# Checks for programs.
AC_PROG_CC

# only for the recursive case
AC_CONFIG_FILES([Makefile
                 lib/Makefile
                 man/Makefile
                 src/Makefile])
AC_OUTPUT
```

Let's package this with the **debmake** command into multiple packages:

- **debhello**: type = **bin**
- **libsharedlib1**: type = **lib**
- **libsharedlib-dev**: type = **dev**

Here, the **-b',libsharedlib1,libsharedlib-dev'** option is used to specify the generated binary packages.

```
 $ cd debhello-2.0
 $ debmake -b',libsharedlib1,libsharedlib-dev'
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="2.0", rev="1"
I: *** start packaging in "debhello-2.0". ***
I: provide debhello_2.0.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-2.0.tar.gz debhello_2.0.orig.tar.gz
I: pwd = "/path/to/debhello-2.0"
I: parse binary package settings: ,libsharedlib1,libsharedlib-dev
I: binary package=debhello Type=bin / Arch=any M-A=foreign
I: binary package=libsharedlib1 Type=lib / Arch=any M-A=same
I: binary package=libsharedlib-dev Type=dev / Arch=any M-A=same
```

```
I: analyze the source tree
I: build_type = Autotools with autoreconf
...
```

The result is similar to Section 8.8 but with more template files.
Let's inspect the notable template files generated.
debian/rules (template file, v=2.0):

```
 $ cat debhello-2.0/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@ --with autoreconf

#override_dh_install:
#       dh_install --list-missing -X.la -X.pyc -X.pyo
```

Let's make this Debian package better as the maintainer.
debian/rules (maintainer version, v=2.0):

```
 $ vim debhello-2.0/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-2.0/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@ --with autoreconf

override_dh_install:
        dh_install --list-missing -X.la
```

debian/control (maintainer version, v=2.0):

```
 $ vim debhello-2.0/debian/control
 ... hack, hack, hack, ...
 $ cat debhello-2.0/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>=11~), dh-autoreconf
Standards-Version: 4.3.0
Homepage: https://salsa.debian.org/debian/debmake-doc

Package: debhello
Architecture: any
Multi-Arch: foreign
Depends: libsharedlib1 (= ${binary:Version}),
         ${misc:Depends},
         ${shlibs:Depends}
Description: example executable package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
```

```
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
 .
 This package provides the executable program.

Package: libsharedlib1
Section: libs
Architecture: any
Multi-Arch: same
Pre-Depends: ${misc:Pre-Depends}
Depends: ${misc:Depends}, ${shlibs:Depends}
Description: example shared library package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
 .
 This package contains the shared library.

Package: libsharedlib-dev
Section: libdevel
Architecture: any
Multi-Arch: same
Depends: libsharedlib1 (= ${binary:Version}), ${misc:Depends}
Description: example development package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
 .
 This package contains the development files.
```

debian/*.install (maintainer version, v=2.0):

```
 $ vim debhello-2.0/debian/debhello.install
 ... hack, hack, hack, ...
 $ cat debhello-2.0/debian/debhello.install
usr/bin/*
usr/share/man/*
 $ vim debhello-2.0/debian/libsharedlib1.install
 ... hack, hack, hack, ...
 $ cat debhello-2.0/debian/libsharedlib1.install
usr/lib/*/*.so.*
 $ vim debhello-2.0/debian/libsharedlib-dev.install
 ... hack, hack, hack, ...
 $ cat debhello-2.0/debian/libsharedlib-dev.install
###usr/lib/*/pkgconfig/*.pc
usr/include
usr/lib/*/*.so
```

Since this upstream source creates the proper auto-generated **Makefile**, there is no need to create **debian/install** and **debian/manpages** files.

There are several other template files under the **debian/** directory. These also need to be updated.

Template files under debian/. (v=2.0):

```
 $ tree debhello-2.0/debian
debhello-2.0/debian
├── README.Debian
├── changelog
├── compat
```

```
├── control
├── copyright
├── debhello.install
├── libsharedlib-dev.install
├── libsharedlib1.install
├── libsharedlib1.symbols
├── patches
│   └── series
├── rules
├── source
│   ├── format
│   └── local-options
└── watch

2 directories, 14 files
```

The rest of the packaging activities are practically the same as the one in Section 8.8.

Here are the generated dependency list of all binary packages.

The generated dependency list of all binary packages (v=2.0):

```
 $ dpkg -f debhello-dbgsym_2.0-1_amd64.deb pre-depends depends recommends con...
Depends: debhello (= 2.0-1)
 $ dpkg -f debhello_2.0-1_amd64.deb pre-depends depends recommends conflicts ...
Depends: libsharedlib1 (= 2.0-1), libc6 (>= 2.2.5)
 $ dpkg -f libsharedlib-dev_2.0-1_amd64.deb pre-depends depends recommends co...
Depends: libsharedlib1 (= 2.0-1)
 $ dpkg -f libsharedlib1-dbgsym_2.0-1_amd64.deb pre-depends depends recommend...
Depends: libsharedlib1 (= 2.0-1)
 $ dpkg -f libsharedlib1_2.0-1_amd64.deb pre-depends depends recommends confl...
Depends: libc6 (>= 2.2.5)
```

8.12 CMake (multi-binary package)

Here is an example of creating a set of Debian binary packages including the executable package, the shared library package, the development file package, and the debug symbol package from a simple C source program using CMake (**CMakeLists.txt** and some files such as **config.h.in**) as its build system. See Section 5.16.2.

Let's assume this upstream tarball to be **debhello-2.1.tar.gz**.

This type of source is meant to be installed as a non-system file, for example, as:

```
 $ tar -xzmf debhello-2.1.tar.gz
 $ cd debhello-2.1
 $ mkdir obj-x86_64-linux-gnu
 $ cd obj-x86_64-linux-gnu
 $ cmake ..
 $ make
 $ make install
```

Let's get the source and make the Debian package.

Download debhello-2.1.tar.gz

```
 $ wget http://www.example.org/download/debhello-2.1.tar.gz
 ...
 $ tar -xzmf debhello-2.1.tar.gz
 $ tree
.
├── debhello-2.1
│   ├── CMakeLists.txt
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
```

```
│   ├── lib
│   │   ├── CMakeLists.txt
│   │   ├── sharedlib.c
│   │   └── sharedlib.h
│   ├── man
│   │   ├── CMakeLists.txt
│   │   └── hello.1
│   └── src
│       ├── CMakeLists.txt
│       ├── config.h.in
│       └── hello.c
└── debhello-2.1.tar.gz

5 directories, 12 files
```

Here, the contents of this source are as follows.

src/hello.c (v=2.1):

```
 $ cat debhello-2.1/src/hello.c
#include "config.h"
#include <stdio.h>
#include <sharedlib.h>
int
main()
{
        printf("Hello, I am " PACKAGE_AUTHOR "!\n");
        sharedlib();
        return 0;
}
```

src/config.h.in (v=2.1):

```
 $ cat debhello-2.1/src/config.h.in
/* name of the package author */
#define PACKAGE_AUTHOR "@PACKAGE_AUTHOR@"
```

lib/sharedlib.c and lib/sharedlib.h (v=2.1):

```
 $ cat debhello-2.1/lib/sharedlib.h
int sharedlib();
 $ cat debhello-2.1/lib/sharedlib.c
#include <stdio.h>
int
sharedlib()
{
        printf("This is a shared library!\n");
        return 0;
}
```

CMakeLists.txt (v=2.1):

```
 $ cat debhello-2.1/CMakeLists.txt
cmake_minimum_required(VERSION 2.8)
project(debhello)
set(PACKAGE_AUTHOR "Osamu Aoki")
add_subdirectory(lib)
add_subdirectory(src)
add_subdirectory(man)
 $ cat debhello-2.1/man/CMakeLists.txt
install(
  FILES ${CMAKE_CURRENT_SOURCE_DIR}/hello.1
  DESTINATION share/man/man1
```

```
)
 $ cat debhello-2.1/src/CMakeLists.txt
# Always define HAVE_CONFIG_H
add_definitions(-DHAVE_CONFIG_H)
# Generate config.h from config.h.in
configure_file(
  "${CMAKE_CURRENT_SOURCE_DIR}/config.h.in"
  "${CMAKE_CURRENT_BINARY_DIR}/config.h"
  )
include_directories("${CMAKE_CURRENT_BINARY_DIR}")
include_directories("${CMAKE_SOURCE_DIR}/lib")

add_executable(hello hello.c)
target_link_libraries(hello sharedlib)
install(TARGETS hello
  RUNTIME DESTINATION bin
)
```

Let's package this with the **debmake** command.

```
 $ cd debhello-2.1
 $ debmake -b',libsharedlib1,libsharedlib-dev'
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="2.1", rev="1"
I: *** start packaging in "debhello-2.1". ***
I: provide debhello_2.1.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-2.1.tar.gz debhello_2.1.orig.tar.gz
I: pwd = "/path/to/debhello-2.1"
I: parse binary package settings: ,libsharedlib1,libsharedlib-dev
I: binary package=debhello Type=bin / Arch=any M-A=foreign
...
```

The result is similar to Section 8.8 but not exactly the same.

Let's inspect the notable template files generated.

debian/rules (template file, v=2.1):

```
 $ cat debhello-2.1/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@

#override_dh_auto_configure:
#       dh_auto_configure -- \
#             -DCMAKE_LIBRARY_ARCHITECTURE="$(DEB_TARGET_MULTIARCH)"
```

Let's make this Debian package better as the maintainer.

debian/rules (maintainer version, v=2.1):

```
 $ vim debhello-2.1/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-2.1/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
```

```
export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed
DEB_HOST_MULTIARCH ?= $(shell dpkg-architecture -qDEB_HOST_MULTIARCH)

%:
        dh $@

override_dh_auto_configure:
        dh_auto_configure -- \
              -DCMAKE_LIBRARY_ARCHITECTURE="$(DEB_HOST_MULTIARCH)"

override_dh_install:
        dh_install --list-missing
```

debian/control (maintainer version, v=2.1):

```
 $ vim debhello-2.1/debian/control
 ... hack, hack, hack, ...
 $ cat debhello-2.1/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: cmake, debhelper (>=11~)
Standards-Version: 4.3.0
Homepage: https://salsa.debian.org/debian/debmake-doc

Package: debhello
Architecture: any
Multi-Arch: foreign
Depends: libsharedlib1 (= ${binary:Version}),
         ${misc:Depends},
         ${shlibs:Depends}
Description: example executable package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
 .
 This package provides the executable program.

Package: libsharedlib1
Section: libs
Architecture: any
Multi-Arch: same
Pre-Depends: ${misc:Pre-Depends}
Depends: ${misc:Depends}, ${shlibs:Depends}
Description: example shared library package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
 .
 This package contains the shared library.

Package: libsharedlib-dev
Section: libdevel
Architecture: any
Multi-Arch: same
Depends: libsharedlib1 (= ${binary:Version}), ${misc:Depends}
Description: example development package
 This is an example package to demonstrate Debian packaging using
```

```
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
 .
 This package contains the development files.
```

debian/*.install (maintainer version, v=2.1):

```
 $ vim debhello-2.1/debian/debhello.install
 ... hack, hack, hack, ...
 $ cat debhello-2.1/debian/debhello.install
usr/bin/*
usr/share/man/*
 $ vim debhello-2.1/debian/libsharedlib1.install
 ... hack, hack, hack, ...
 $ cat debhello-2.1/debian/libsharedlib1.install
usr/lib/*/*.so.*
 $ vim debhello-2.1/debian/libsharedlib-dev.install
 ... hack, hack, hack, ...
 $ cat debhello-2.1/debian/libsharedlib-dev.install
###usr/lib/*/pkgconfig/*.pc
usr/include
usr/lib/*/*.so
```

This upstream CMakeList.txt needs to be patched to cope with the multiarch path.

debian/patches/* (maintainer version, v=2.1):

```
 ... hack, hack, hack, ...
 $ cat debhello-2.1/debian/libsharedlib1.symbols
libsharedlib.so.1 libsharedlib1 #MINVER#
 sharedlib@Base 2.1
```

Since this upstream source creates the proper auto-generated **Makefile**, there is no need to create **debian/install** and **debian/manpages** files.

There are several other template files under the **debian/** directory. These also need to be updated.

Template files under debian/. (v=2.1):

```
 $ tree debhello-2.1/debian
debhello-2.1/debian
├── README.Debian
├── changelog
├── compat
├── control
├── copyright
├── debhello.install
├── libsharedlib-dev.install
├── libsharedlib1.install
├── libsharedlib1.symbols
├── patches
│   ├── 000-cmake-multiarch.patch
│   └── series
├── rules
├── source
│   ├── format
│   └── local-options
└── watch

2 directories, 15 files
```

The rest of the packaging activities are practically the same as the one in Section 8.8.

Here are the generated dependency list of all binary packages.

The generated dependency list of all binary packages (v=2.1):

```
 $ dpkg -f debhello-dbgsym_2.1-1_amd64.deb pre-depends depends recommends con...
Depends: debhello (= 2.1-1)
 $ dpkg -f debhello_2.1-1_amd64.deb pre-depends depends recommends conflicts ...
Depends: libsharedlib1 (= 2.1-1), libc6 (>= 2.2.5)
 $ dpkg -f libsharedlib-dev_2.1-1_amd64.deb pre-depends depends recommends co...
Depends: libsharedlib1 (= 2.1-1)
 $ dpkg -f libsharedlib1-dbgsym_2.1-1_amd64.deb pre-depends depends recommend...
Depends: libsharedlib1 (= 2.1-1)
 $ dpkg -f libsharedlib1_2.1-1_amd64.deb pre-depends depends recommends confl...
Depends: libc6 (>= 2.2.5)
```

8.13 Internationalization

Here is an example of updating the simple upstream C source **debhello-2.0.tar.gz** presented in Section 8.11 for internationalization (i18n) and creating the updated upstream C source **debhello-2.0.tar.gz**.

In the real situation, the package should already be internationalized. So this example is educational for you to understand how this internationalization is implemented.

> Tip
>
> 
>
> The routine maintainer activity for the i18n is simply to add translation po files reported to you via the Bug Tracking System (BTS) to the **po/** directory and to update the language list in the **po/LINGUAS** file.

Let's get the source and make the Debian package.

Download debhello-2.0.tar.gz (i18n)

```
 $ wget http://www.example.org/download/debhello-2.0.tar.gz
 ...
 $ tar -xzmf debhello-2.0.tar.gz
 $ tree
.
├── debhello-2.0
│   ├── Makefile.am
│   ├── configure.ac
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── lib
│   │   ├── Makefile.am
│   │   ├── sharedlib.c
│   │   └── sharedlib.h
│   ├── man
│   │   ├── Makefile.am
│   │   └── hello.1
│   └── src
│       ├── Makefile.am
│       └── hello.c
└── debhello-2.0.tar.gz

5 directories, 12 files
```

Internationalize this source tree with the **gettextize** command and remove files auto-generated by Autotools.

run gettextize (i18n):

```
 $ cd debhello-2.0
 $ gettextize
Creating po/ subdirectory
Creating build-aux/ subdirectory
Copying file ABOUT-NLS
Copying file build-aux/config.rpath
Not copying intl/ directory.
Copying file po/Makefile.in.in
Copying file po/Makevars.template
Copying file po/Rules-quot
Copying file po/boldquot.sed
Copying file po/en@boldquot.header
Copying file po/en@quot.header
Copying file po/insert-header.sin
Copying file po/quot.sed
Copying file po/remove-potcdate.sin
Creating initial po/POTFILES.in
Creating po/ChangeLog
Creating directory m4
Copying file m4/gettext.m4
Copying file m4/iconv.m4
Copying file m4/lib-ld.m4
Copying file m4/lib-link.m4
Copying file m4/lib-prefix.m4
Copying file m4/nls.m4
Copying file m4/po.m4
Copying file m4/progtest.m4
Creating m4/ChangeLog
Updating Makefile.am (backup is in Makefile.am~)
Updating configure.ac (backup is in configure.ac~)
Creating ChangeLog

Please use AM_GNU_GETTEXT([external]) in order to cause autoconfiguration
to look for an external libintl.

Please create po/Makevars from the template in po/Makevars.template.
You can then remove po/Makevars.template.

Please fill po/POTFILES.in as described in the documentation.

Please run 'aclocal' to regenerate the aclocal.m4 file.
You need aclocal from GNU automake 1.9 (or newer) to do this.
Then run 'autoconf' to regenerate the configure file.

You will also need config.guess and config.sub, which you can get from the CV...
of the 'config' project at http://savannah.gnu.org/. The commands to fetch th...
are
$ wget 'http://savannah.gnu.org/cgi-bin/viewcvs/*checkout*/config/config/conf...
$ wget 'http://savannah.gnu.org/cgi-bin/viewcvs/*checkout*/config/config/conf...

You might also want to copy the convenience header file gettext.h
from the /usr/share/gettext directory into your package.
It is a wrapper around <libintl.h> that implements the configure --disable-nl...
option.

Press Return to acknowledge the previous 6 paragraphs.
 $ rm -rf m4 build-aux *~
```

Let's check generated files under the **po/** directory.
files in po (i18n):

```
 $ ls -l po
/build/debmake-doc-1.14/debhello-2.0-pkg2/step151.cmd: line 2: SOURCE_DATE_EP...
```

```
total 60
-rw-r--r-- 1 pbuilder pbuilder   494 May 25 20:41 ChangeLog
-rw-r--r-- 1 pbuilder pbuilder 17577 May 25 20:41 Makefile.in.in
-rw-r--r-- 1 pbuilder pbuilder  3376 May 25 20:41 Makevars.template
-rw-r--r-- 1 pbuilder pbuilder    59 May 25 20:41 POTFILES.in
-rw-r--r-- 1 pbuilder pbuilder  2203 May 25 20:41 Rules-quot
-rw-r--r-- 1 pbuilder pbuilder   217 May 25 20:41 boldquot.sed
-rw-r--r-- 1 pbuilder pbuilder  1337 May 25 20:41 en@boldquot.header
-rw-r--r-- 1 pbuilder pbuilder  1203 May 25 20:41 en@quot.header
-rw-r--r-- 1 pbuilder pbuilder   672 May 25 20:41 insert-header.sin
-rw-r--r-- 1 pbuilder pbuilder   153 May 25 20:41 quot.sed
-rw-r--r-- 1 pbuilder pbuilder   432 May 25 20:41 remove-potcdate.sin
```

Let's update the **configure.ac** by adding "**AM_GNU_GETTEXT([external])**", etc..
configure.ac (i18n):

```
 $ vim configure.ac
 ... hack, hack, hack, ...
 $ cat configure.ac
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ([2.69])
AC_INIT([debhello],[2.2],[foo@example.org])
AC_CONFIG_SRCDIR([src/hello.c])
AC_CONFIG_HEADERS([config.h])
echo "Standard customization chores"
AC_CONFIG_AUX_DIR([build-aux])

AM_INIT_AUTOMAKE([foreign])

# Set default to --enable-shared --disable-static
LT_INIT([shared disable-static])

# find the libltdl sources in the libltdl sub-directory
LT_CONFIG_LTDL_DIR([libltdl])

# choose one
LTDL_INIT([recursive])
#LTDL_INIT([subproject])
#LTDL_INIT([nonrecursive])

# Add #define PACKAGE_AUTHOR ... in config.h with a comment
AC_DEFINE(PACKAGE_AUTHOR, ["Osamu Aoki"], [Define PACKAGE_AUTHOR])
# Checks for programs.
AC_PROG_CC

# desktop file support required
AM_GNU_GETTEXT_VERSION([0.19.3])
AM_GNU_GETTEXT([external])

# only for the recursive case
AC_CONFIG_FILES([Makefile
                 po/Makefile.in
                 lib/Makefile
                 man/Makefile
                 src/Makefile])
AC_OUTPUT
```

Let's create the **po/Makevars** file from the **po/Makevars.template** file.
po/Makevars (i18n):

```
 ... hack, hack, hack, ...
 $ diff -u po/Makevars.template po/Makevars
```

```
--- po/Makevars.template        2019-03-26 17:03:20.165623558 +0000
+++ po/Makevars 2019-03-26 17:03:20.245621814 +0000
@@ -18,14 +18,14 @@
 # or entity, or to disclaim their copyright.  The empty string stands for
 # the public domain; in this case the translators are expected to disclaim
 # their copyright.
-COPYRIGHT_HOLDER = Free Software Foundation, Inc.
+COPYRIGHT_HOLDER = Osamu Aoki <osamu@debian.org>

 # This tells whether or not to prepend "GNU " prefix to the package
 # name that gets inserted into the header of the $(DOMAIN).pot file.
 # Possible values are "yes", "no", or empty.  If it is empty, try to
 # detect it automatically by scanning the files in $(top_srcdir) for
 # "GNU packagename" string.
-PACKAGE_GNU =
+PACKAGE_GNU = no

 # This is the email address or URL to which the translators shall report
 # bugs in the untranslated strings:
 $ rm po/Makevars.template
```

Let's update C sources for the i18n version by wrapping strings with **_(…)**.
src/hello.c (i18n):

```
 ... hack, hack, hack, ...
 $ cat src/hello.c
#include "config.h"
#include <stdio.h>
#include <sharedlib.h>
#define _(string) gettext (string)
int
main()
{
        printf(_("Hello, I am " PACKAGE_AUTHOR "!\n"));
        sharedlib();
        return 0;
}
```

lib/sharedlib.c (i18n):

```
 ... hack, hack, hack, ...
 $ cat lib/sharedlib.c
#include <stdio.h>
#define _(string) gettext (string)
int
sharedlib()
{
        printf(_("This is a shared library!\n"));
        return 0;
}
```

The new **gettext** (v=0.19) can handle the i18n version of the desktop file directly.
data/hello.desktop.in (i18n):

```
 $ fgrep -v '[ja]=' data/hello.desktop > data/hello.desktop.in
 $ rm data/hello.desktop
 $ cat data/hello.desktop.in
[Desktop Entry]
Name=Hello
Comment=Greetings
Type=Application
Keywords=hello
```

```
Exec=hello
Terminal=true
Icon=hello.png
Categories=Utility;
```

Let's list the input files to extract translatable strings in **po/POTFILES.in**.
po/POTFILES.in (i18n):

```
 ... hack, hack, hack, ...
 $ cat po/POTFILES.in
src/hello.c
lib/sharedlib.c
data/hello.desktop.in
```

Here is the updated root **Makefile.am** with **po** added to the **SUBDIRS** environment variable.
Makefile.am (i18n):

```
 $ cat Makefile.am
# recursively process `Makefile.am` in SUBDIRS
SUBDIRS = po lib src man

ACLOCAL_AMFLAGS = -I m4

EXTRA_DIST = build-aux/config.rpath m4/ChangeLog
```

Let's make a translation template file, **debhello.pot**.
po/debhello.pot (i18n):

```
 $ xgettext -f po/POTFILES.in -d debhello -o po/debhello.pot -k_
 $ cat po/debhello.pot
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2019-03-26 17:03+0000\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#: src/hello.c:8
#, c-format
msgid "Hello, I am "
msgstr ""

#: lib/sharedlib.c:6
#, c-format
msgid "This is a shared library!\n"
msgstr ""

#: data/hello.desktop.in:3
msgid "Hello"
msgstr ""
```

```
#: data/hello.desktop.in:4
msgid "Greetings"
msgstr ""

#: data/hello.desktop.in:6
msgid "hello"
msgstr ""

#: data/hello.desktop.in:9
msgid "hello.png"
msgstr ""
```

Let's add a translation for French.
po/LINGUAS and po/fr.po (i18n):

```
 $ echo 'fr' > po/LINGUAS
 $ cp po/debhello.pot po/fr.po
 $ vim po/fr.po
 ... hack, hack, hack, ...
 $ cat po/fr.po
# SOME DESCRIPTIVE TITLE.
# This file is put in the public domain.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
msgid ""
msgstr ""
"Project-Id-Version: debhello 2.2\n"
"Report-Msgid-Bugs-To: foo@example.org\n"
"POT-Creation-Date: 2015-03-01 20:22+0900\n"
"PO-Revision-Date: 2015-02-21 23:18+0900\n"
"Last-Translator: Osamu Aoki <osamu@debian.org>\n"
"Language-Team: French <LL@li.org>\n"
"Language: ja\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: src/hello.c:34
#, c-format
msgid "Hello, my name is %s!\n"
msgstr "Bonjour, je m'appelle %s!\n"

#: lib/sharedlib.c:29
#, c-format
msgid "This is a shared library!\n"
msgstr "Ceci est une bibliothèque partagée!\n"

#: data/hello.desktop.in:3
msgid "Hello"
msgstr ""

#: data/hello.desktop.in:4
msgid "Greetings"
msgstr "Salutations"

#: data/hello.desktop.in:6
msgid "hello"
msgstr ""

#: data/hello.desktop.in:9
msgid "hello.png"
msgstr ""
```

The packaging activities are practically the same as the one in Section 8.11.
You can find more i18n examples in Section 8.14 for

- the POSIX shell script with Makefile (v=3.0),
- the Python3 script with distutils (v=3.1),
- the C source with Makefile.in + configure (v=3.2),
- the C source with Autotools (v=3.3), and
- the C source with CMake (v=3.4).

8.14 Details

Actual details of the examples presented and their variants can be obtained by the following.

How to get details

```
$ apt-get source debmake-doc
$ sudo apt-get install devscripts build-essentials
$ cd debmake-doc*
$ sudo apt-get build-dep ./
$ make
```

Each directory with the **-pkg[0-9]** suffix contains the Debian packaging example.

- emulated console command line activity log: the **.log** file
- emulated console command line activity log (short): the **.slog** file
- snapshot source tree image after the **debmake** command: the **debmake** directory
- snapshot source tree image after proper packaging: the **packge** directory
- snapshot source tree image after the **debuild** command: the **test** directory

Appendix A

debmake(1) manpage

A.1 NAME

debmake - program to make a Debian source package

A.2 SYNOPSIS

debmake [**-h**] [**-c** | **-k**] [**-n** | **-a** *package-version***.orig.tar.gz** | **-d** | **-t**] [**-p** *package*] [**-u** *version*] [**-r** *revision*] [**-z** *extension*] [**-b** "*binarypackage, …*]" [**-e** *foo@example.org*] [**-f** "*firstname lastname*"] [**-i** "*buildtool*" | **-j**] [**-l** *license_file*] [**-m**] [**-o** *file*] [**-q**] [**-s**] [**-v**] [**-w** "*addon, …*"] [**-x** [01234]] [**-y**] [**-L**] [**-P**] [**-T**]

A.3 DESCRIPTION

debmake helps to build a Debian package from the upstream source. Normally, this is done as follows:

- The upstream tarball is downloaded as the *package-version***.tar.gz** file.
- It is untarred to create many files under the *package-version/* directory.
- debmake is invoked in the *package-version/* directory, possibly without any arguments.
- Files in the *package-version*/**debian/** directory are manually adjusted.
- **dpkg-buildpackage** (usually from its wrapper **debuild** or **pdebuild**) is invoked in the *package-version/* directory to make Debian packages.

Make sure to protect the arguments of the **-b**, **-f**, **-l**, and **-w** options from shell interference by quoting them properly.

A.3.1 optional arguments:

-h, --help show this help message and exit.

-c, --copyright scan source for copyright+license text and exit.

- **-c**: simple output style
- **-cc**: normal output style (similar to the **debian/copyright** file)
- **-ccc**: debug output style

-k, --kludge compare the **debian/copyright** file with the source and exit.

The **debian/copyright** file must be organized to list the generic file patterns before the specific exceptions.

- **-k**: basic output style
- **-kk**: verbose output style

-n, --native make a native Debian source package without **.orig.tar.gz**. This makes a "**3.0 (native)**" format package.

If you are thinking of packaging a Debian-specific source tree with **debian/*** in it into a native Debian package, please think otherwise. You can use the "**debmake -d -i debuild**" or "**debmake -t -i debuild**" commands to make a "**3.0 (quilt)**" format non-native Debian package. The only difference is that the **debian/changelog** file must use the non-native version scheme: *version-revision*. The non-native package is more friendly to downstream distributions.

-a *package*-*version*.tar.gz, --archive *package*-*version*.tar.gz use the upstream source tarball directly. (**-p**, **-u**, **-z**: overridden)

The upstream tarball may be specified as *package_version***.orig.tar.gz** and **tar.gz**. For other cases, it may be **tar.bz2**, or **tar.xz**.

If the specified upstream tarball name contains uppercase letters, the Debian package name is generated by converting them to lowercase letters.

If the specified argument is the URL (http://, https://, or ftp://) to the upstream tarball, the upstream tarball is downloaded from the URL using **wget** or **curl**.

-d, --dist run the "make dist" command equivalents first to generate the upstream tarball and use it.

The "**debmake -d**" command is designed to run in the *package/* directory hosting the upstream VCS with the build system supporting the "**make dist**" command equivalents. (automake/autoconf, Python distutils, …)

-t, --tar run the "**tar**" command to generate the upstream tarball and use it.

The "**debmake -t**" command is designed to run in the *package/* directory hosting the upstream VCS. Unless you provide the upstream version with the **-u** option or with the **debian/changelog** file, a snapshot upstream version is generated in the **0~%y%m%d%H%M** format, e.g., *0~1403012359*, from the UTC date and time. The generated tarball excludes the **debian/** directory found in the upstream VCS. (It also excludes typical VCS directories: **.git/ .hg/ .svn/ .CVS/**.)

-p *package*, --package *package* set the Debian package name.

-u *version*, --upstreamversion *version* set the upstream package version.

-r *revision*, --revision *revision* set the Debian package revision.

-z *extension*, --targz *extension* set the tarball type, *extension*=(**tar.gz**|**tar.bz2**|**tar.xz**). (alias: **z**, **b**, **x**)

-b *"binarypackage[:type],…", --binaryspec "binarypackage[:type],…"* set the binary package specs by a comma separated list of *binarypackage:type* pairs, e.g., in the full form "**foo:bin,foo-doc:doc,libfoo1:lib,libfoo-dev:dev**" or in the short form, "**-doc,libfoo1,libfoo-dev**".

Here, *binarypackage* is the binary package name, and the optional *type* is chosen from the following *type* values:

- **bin**: C/C++ compiled ELF binary code package (any, foreign) (default, alias: **""**, i.e., *null-string*)
- **data**: Data (fonts, graphics, …) package (all, foreign) (alias: **da**)
- **dev**: Library development package (any, same) (alias: **de**)
- **doc**: Documentation package (all, foreign) (alias: **do**)
- **lib**: Library package (any, same) (alias: **l**)
- **perl**: Perl script package (all, foreign) (alias: **pl**)
- **python**: Python script package (all, foreign) (alias: **py**)
- **python3**: Python3 script package (all, foreign) (alias: **py3**)
- **ruby**: Ruby script package (all, foreign) (alias: **rb**)
- **script**: Shell script package (all, foreign) (alias: **sh**)

The pair values in the parentheses, such as (any, foreign), are the **Architecture** and **Multi-Arch** stanza values set in the **debian/control** file.

In many cases, the **debmake** command makes good guesses for *type* from *binarypackage*. If *type* is not obvious, *type* is set to **bin**. For example, **libfoo** sets *type* to **lib**, and **font-bar** sets *type* to **data**, …

If the source tree contents do not match settings for *type*, the **debmake** command warns you.

-e *foo@example.org*, --email *foo@example.org* set e-mail address.

The default is taken from the value of the environment variable **$DEBEMAIL**.

-f *"firstname lastname"*, --fullname *"firstname lastname"* set the fullname.

The default is taken from the value of the environment variable **$DEBFULLNAME**.

-i *"buildtool"*, --invoke *"buildtool"* invoke "*buildtool*" at the end of execution. *buildtool* may be "**dpkg-buildpackage**", "**debuild**", "**pdebuild**", "**pdebuild --pbuilder cowbuilder**", etc.

The default is not to execute any program.

Setting this option automatically sets the **--local** option.

-j, --judge run **dpkg-depcheck** to judge build dependencies and identify file paths. Log files are in the parent directory.

- *package***.build-dep.log**: Log file for **dpkg-depcheck**.
- *package***.install.log**: Log file recording files in the **debian/tmp** directory.

-l *"license_file,…"*, --license *"license_file,…"* add formatted license text to the end of the **debian/copyright** file holding license scan results.

The default is to add **COPYING** and **LICENSE**, and *license_file* needs to list only the additional file names all separated by "**,**".

-m, --monoarch force packages to be non-multiarch.

-o *file*, --option *file* read optional parameters from *file*. (This is not for everyday use.)

The content of *file* is sourced as the Python3 code at the end of **para.py**. For example, the package description can be specified by the following file.

```
para['desc'] = 'program short description'
para['desc_long'] = '''\
 program long description which you wish to include.
 .
 Empty line is space + .
 You keep going on ...
'''
```

-q, --quitearly quit early before creating files in the **debian/** directory.

-s, --spec use upstream spec (setup.py for Python, etc.) for the package description.

-v, --version show version information.

-w *"addon,…"*, --with *"addon,…"* add extra arguments to the **--with** option of the **dh**(1) command as *addon* in **debian/rules**.

The *addon* values are listed all separated by "**,**", e.g., "**-w "python2,autoreconf""**.

For Autotools based packages, setting **autoreconf** as *addon* forces running "**autoreconf -i -v -f**" for every package building. Otherwise, **autotools-dev** as *addon* is used as the default.

For Autotools based packages, if they install Python programs, **python2** as *addon* is needed for packages with "**compat < 9**" since this is non-obvious. But for **setup.py** based packages, **python2** as *addon* is not needed since this is obvious and it is automatically set for the **dh**(1) command by the **debmake** command when it is required.

-x *n*, --extra *n* generate configuration files as templates. (Please note **debian/changelog**, **debian/control**, **debian/copyright**, and **debian/rules** are bare minimum configuration files to build a Debian binary package.)

The number *n* determines which configuration templates are generated.

- **-x0**: bare minimum configuration files. (default option if any of bare minimum configuration files already exist)

- **-x1**: all **-x0** files + desirable configuration files for the single binary package. (default option for the single binary package if none of bare minimum configuration files exist)
- **-x2**: all **-x1** files + desirable configuration files for the multi binary package. (default option for the multi binary package if none of bare minimum configuration files exist)
- **-x3**: all **-x2** files + unusual configuration template files. Unusual configuration template files are generated with the extra **.ex** suffix to ease their removal. To use these as configuration files, rename their file names to ones without the **.ex** suffix.
- **-x4**: all **-x3** files + copyright file examples.

-y, --yes "force yes" for all prompts. (without option: "ask [Y/n]"; doubled option: "force no")

-L, --local generate configuration files for the local package to fool **lintian**(1) checks.

-P, --pedantic pedantically check auto-generated files.

-T, --tutorial output tutorial comment lines in template files.

A.4 EXAMPLES

For a well behaving source, you can build a good-for-local-use installable single Debian binary package easily with one command. Test install of such a package generated in this way offers a good alternative to the traditional "**make install**" command installing into the **/usr/local** directory since the Debian package can be removed cleanly by the "**dpkg -P** …" command. Here are some examples of how to build such test packages. (These should work in most cases. If the **-d** option does not work, try the **-t** option instead.)

For a typical C program source tree packaged with autoconf/automake:

- **debmake -d -i debuild**

For a typical Python module source tree:

- **debmake -s -d -b":python" -i debuild**

For a typical Python module in the *package-version***.tar.gz** archive:

- **debmake -s -a *package-version*.tar.gz -b":python" -i debuild**

For a typical Perl module in the *package-version***.tar.gz** archive:

- **debmake -a *package-version*.tar.gz -b":perl" -i debuild**

A.5 HELPER PACKAGES

Packaging may require installation of some additional specialty helper packages.

- Python3 programs may require the **dh-python** package.
- The Autotools (Autoconf + Automake) build system may require **autotools-dev** or **dh-autoreconf** package.
- Ruby programs may require the **gem2deb** package.
- Java programs may require the **javahelper** package.
- Gnome programs may require the **gobject-introspection** package.
- etc.

A.6 CAVEAT

debmake is meant to provide template files for the package maintainer to work on. Comment lines started by **#** contain the tutorial text. You must remove or edit such comment lines before uploading to the Debian archive.

The license extraction and assignment process involves a lot of heuristics; it may fail in some cases. It is highly recommended to use other tools such as **licensecheck** from the **devscripts** package in conjunction with **debmake**.

There are some limitations for what characters may be used as a part of the Debian package. The most notable limitation is the prohibition of uppercase letters in the package name. Here is a summary as a set of regular expressions:

- Upstream package name (**-p**): [-+.a-z0-9]{2,}
- Binary package name (**-b**): [-+.a-z0-9]{2,}
- Upstream version (**-u**): [0-9][-+.:~a-z0-9A-Z]*
- Debian revision (**-r**): [0-9][+.~a-z0-9A-Z]*

See the exact definition in Chapter 5 - Control files and their fields in the "Debian Policy Manual".

debmake assumes relatively simple packaging cases. So all programs related to the interpreter are assumed to be "**Architecture: all**". This is not always true.

A.7 DEBUG

Please report bugs to the **debmake** package using the **reportbug** command.

The character set in the environment variable **$DEBUG** determines the logging output level.

- **i**: print information
- **p**: list all global parameters
- **d**: list parsed parameters for all binary packages
- **f**: input filename for the copyright scan
- **y**: year/name split of copyright line
- **s**: line scanner for format_state
- **b**: content_state scan loop: begin-loop
- **m**: content_state scan loop: after regex match
- **e**: content_state scan loop: end-loop
- **c**: print copyright section text
- **l**: print license section text
- **a**: print author/translator section text
- **k**: sort key for debian/copyright stanza
- **n**: scan result of debian/copyright ("**debmake -k**")

Use this as:

```
$ DEBUG=pdfbmeclak debmake ...
```

See README.developer in the source for more.

A.8 AUTHOR

Copyright © 2014-2017 Osamu Aoki <osamu@debian.org>

A.9 LICENSE

Expat License

A.10 SEE ALSO

The **debmake-doc** package provides the “Guide for Debian Maintainers” in plain text, HTML and PDF formats under the **/usr/share/doc/debmake-doc/** directory.

See also **dpkg-source**(1), **deb-control**(5), **debhelper**(7), **dh**(1), **dpkg-buildpackage**(1), **debuild**(1), **quilt**(1), **dpkg-depcheck**(1), **pdebuild**(1), **pbuilder**(8), **cowbuilder**(8), **gbp-buildpackage**(1), **gbp-pq**(1), and **git-pbuilder**(1) manpages.