

debian

Debian メンテナー用ガイド

青木修 [FAMILY Given]

October 8, 2026

Debian メンテナー用ガイド
by 青木修 [FAMILY Given]

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.

本ガイドの最新版は以下から入手できます:

- 「debmake-doc パッケージ」中と、
- 「Debian Debian 開発者向けマニュアル」のウエッブサイト上

Contents

1 序文 **1**

2 Overview **3**

3 Prerequisites **5**
3.1 People around Debian . . . 5
3.2 How to contribute . . . 5
3.3 Debian における社会ダイナミクス . . . 6
3.4 Technical reminders . . . 6
3.5 Debian documentation . . . 7
3.6 Help resources . . . 8
3.7 Archive situation . . . 8
3.8 Contribution approaches . . . 9
3.9 Novice contributor and maintainer . . . 10

4 Tool Setups **12**
4.1 Email setup . . . 12
4.2 **mc** setup . . . 13
4.3 **git** setup . . . 13
4.4 **quilt** setup . . . 13
4.5 **openPGP** setup . . . 14
4.6 **devscripts** setup . . . 15
4.7 **sbuild** setup . . . 15
4.8 Persistent chroot setup . . . 17
4.9 **gbp** setup . . . 18
4.10 HTTP proxy . . . 18
4.11 Private Debian repository . . . 18
4.12 Virtual machines . . . 18
4.13 Local network with virtual machines . . . 18

5 Simple packaging **19**
5.1 Packaging tarball . . . 19
5.2 Big picture . . . 19
5.3 What is debmake? . . . 20
5.4 What is debuild? . . . 21
5.5 Step 1: Get the upstream source . . . 21
5.6 Step 2: Generate template files with **debmake** . . . 22
5.7 Step 3: Modification to the template files . . . 27
5.8 Step 4: Building package with **debuild** . . . 29
5.9 Step 3 (alternatives): Modification to the upstream source . . . 31
5.10 Patch by 「**diff -u**」approach . . . 32
5.11 Patch by **dquilt** approach . . . 33
5.12 Patch by 「**dpkg-source --auto-commit**」approach . . . 35

6 Basics for packaging **38**
6.1 Packaging workflow . . . 38
6.2 **debhelper** package . . . 40
6.3 パッケージ名とバージョン . . . 41
6.4 ネイティブ Debian パッケージ . . . 42
6.5 **debian/rules** file . . . 42
6.6 **debian/control** file . . . 43
6.7 **debian/changelog** file . . . 44
6.8 **debian/copyright** file . . . 45
6.9 **debian/patches/*** files . . . 45

6.10 **debian/source/include-binaries** file . . . 46
6.11 **debian/watch** file . . . 46
6.12 **debian/upstream/signing-key.asc** file . . . 46
6.13 **debian/salsa-ci.yml** file . . . 47
6.14 Other **debian/*** files . . . 47

7 Quality of packaging **52**
7.1 Reformat **debian/*** files with **wrap-and-sort** . . . 52
7.2 Validate **debian/*** files with **debputy** . . . 52
7.3 Check packaging with **cme** . . . 52

8 Sanitization of the source **53**
8.1 Fix with **Files-Excluded** . . . 53
8.2 Fix with 「**debian/rules clean**」 . . . 53
8.3 Fix with **extend-diff-ignore** . . . 54
8.4 Fix with **tar-ignore** . . . 54
8.5 Fix with 「**git clean -dfx**」 . . . 55

9 More on packaging **56**
9.1 Package customization . . . 56
9.2 Customized **debian/rules** . . . 56
9.3 Variables for **debian/rules** . . . 57
9.4 アップストリームソフトウェアの新版更新 . . . 57
9.5 Manage patch queue with **dquilt** . . . 58
9.6 Build commands . . . 58
9.7 Note on **sbuild** . . . 58
9.8 Special build cases . . . 59
9.9 Upload **orig.tar.xz** . . . 59
9.10 スキップされたアップロード . . . 60
9.11 Bug reports . . . 60

10 上級パッケージング **62**
10.1 Historical perspective . . . 62
10.2 Current trends . . . 62
10.3 Note on build system . . . 63
10.4 Continuous integration . . . 63
10.5 Bootstrapping . . . 63
10.6 Compiler hardening . . . 64
10.7 Reproducible build . . . 64
10.8 Substvar . . . 64
10.9 Library package . . . 65
10.10マルチアーチ . . . 66
10.11Split of a Debian binary package . . . 66
10.12Package split scenario and examples . . . 67
10.13Multiarch library path . . . 67
10.14Multiarch header file path . . . 68
10.15Multiarch ***.pc** file path . . . 68
10.16Library symbols . . . 68
10.17Library package name . . . 69
10.18Library transition . . . 70
10.19binNMU safe . . . 70
10.20Debugging information . . . 71
10.21**-dbgsym** package . . . 71
10.22**debconf** . . . 71

11 Packaging with git **73**
11.1 Salsa repository . . . 74
11.2 Salsa account setup . . . 74
11.3 Salsa CI service . . . 74
11.4 Branch names . . . 74
11.5 Patch unapplied Git repository . . . 75
11.6 Patch by 「**gbp-pq**」approach . . . 75
11.7 Manage patch queue with **gbp-pq** . . . 75
11.8 **gbp import-dscs --debsnap** . . . 76
11.9 Note on **gbp** . . . 76
11.10 The Git repository browser . . . 77
11.11 Git commit history organization . . . 77
11.12 **Quasi-native** Debian packaging . . . 78
11.13 Patch applied Git repository . . . 78
11.14 Note on **dgit** . . . 79

12 Tips **81**
12.1 Build under UTF-8 . . . 81
12.2 UTF-8 変換 . . . 81
12.3 Hints for Debugging . . . 81

13 Tool usages **84**
13.1 **debdiff** . . . 84
13.2 **dget** . . . 84
13.3 **mk-origtargz** . . . 85
13.4 **origtargz** . . . 85
13.5 **git deborig** . . . 85
13.6 **dpkg-source -b** . . . 85
13.7 **dpkg-source -x** . . . 85
13.8 **debc** . . . 85
13.9 **bts** . . . 86
13.10 **dpkg-depcheck** . . . 86

14 More Examples **87**
14.1 Cherry-pick templates . . . 87
14.2 No Makefile (shell, CLI) . . . 89
14.3 Makefile (shell, CLI) . . . 96
14.4 pyproject.toml (Python3, CLI) . . . 99
14.5 Makefile (shell, GUI) . . . 105
14.6 pyproject.toml (Python3, GUI) . . . 108
14.7 Makefile (single-binary package) . . . 111
14.8 Makefile.in + configure (single-binary package) . . . 114
14.9 Autotools (single-binary package) . . . 118
14.10 CMake (single-binary package) . . . 122
14.11 Autotools (multi-binary package) . . . 126
14.12 CMake (multi-binary package) . . . 132
14.13 Internationalization . . . 138
14.14 Details . . . 144

15 debmake(1) manpage **145**
15.1 NAME . . . 145
15.2 SYNOPSIS . . . 145
15.3 DESCRIPTION . . . 145
15.4 Positional arguments . . . 146
15.5 Options . . . 146
15.6 EXAMPLES . . . 147
15.7 HELPER PACKAGES . . . 148
15.8 CAVEAT . . . 148
15.9 DEBUG . . . 149

15.10 AUTHOR 149
15.11 LICENSE 149
15.12 SEE ALSO 149

16 debmake options **150**
16.1 Shortcut option (**-i**) 150
16.2 **debmake -b** 150
16.3 **debmake -B** 151
16.4 **debmake -x** 151

Abstract

This “Guide for Debian Maintainers”(2026-10-08) tutorial guide describes the building of the Debian package to ordinary Debian users and prospective developers using the **debmake** command.

本ガイドは現代的なパケージングのスタイルにフォーカスし多くの単純な例とともに提供されます。

- POSIX シェルスクリプトのパッケージング
- Python3 スクリプトのパッケージング
- Makefile/Autotools/CMake を使う C
- 共有ライブラリー等を含む複数バイナリーパッケージ

本「Debian メンテナー用ガイド」は「Debian 新メンテナガイド」の後継書と見なせます。

Chapter 1

序文

もしあなたが少々経験ある Debian ユーザー[1] なら、以下のような状況に出会われたことがあるでしょう:

- 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 want to create a Debian package to fulfill these needs and 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, as it is a large volunteer organization with a rich history. Debian has also developed an extensive array of packaging and archive maintenance tools to build consistent sets of binary packages that address many technical objectives:

- packages have clearly specified package dependencies and patches and build correctly from scratch in a clean build environment (「項 6.6」, 「項 6.9」, 「項 4.7」)
- packages build across many architectures (「項 9.3」)
- builds are reproducible (「項 10.7」)
- multiarch is supported (「項 10.10」)
- bootstrapping new architectures is possible (「項 10.5」)
- builds use specific compiler flags to harden security (「項 10.6」)
- packages are split optimally into multiple binary packages (「項 10.11」)
- library names and contents are managed to ensure smooth transitions on upgrades (「項 10.18」)
- installations use interactive prompts correctly (if at all) (「項 10.22」)
- continuous integration is used to ensure quality (「項 10.4」)
- …

These factors can be overwhelming for many new prospective Debian maintainers. This guide aims to provide entry points to help them get started. It covers 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.

1Unix プログラミングに関して一定の知識は必要ですが、エキスパートである必要はありません。「Debian レファレンス」から基本的な Debian システムの扱い方は学べます。そこでは Unix プログラミングに関する学習の指針も示されています。

2If you're not interested in sharing the Debian package, you can address your local needs by compiling and installing the fixed upstream source package into **/usr/local/**.

- Tips for making the Debian package with minimal effort.
- Examples of making Debian packages in typical scenarios.

The author recognized the limitations of updating the original「New Maintainers' Guide」with the **dh-make** package and decided to create an alternative tool with accompanying documentation to address modern requirements such as multi-arch. This resulted in the **debmake** package, initially released as version 4.0 in 2013. The current **debmake** version is 5.1.5. It comes with this updated「Guide for Debian Maintainers」in the **debmake-doc** package (version: 1.31). (In 2016, **dh-make** was ported from Perl to Python with updated features.)

Many chores and tips have been integrated into the **debmake** command allowing this guide to be terse. This guide also offers many packaging examples for you to get started.

注意



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. While some may seem irrelevant to you, please be patient. Certain corner cases are omitted, and some topics are only covered through external references. These are intentional choices to keep this guide simple and maintainable.

Chapter 2

Overview

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

```
[base_dir] $ tar --xz -xvf package-1.0.tar.xz
[base_dir] $ cd package-1.0
[package-1.0] $ debmake
   ... Make manual adjustments of generated configuration files
[package-1.0] $ 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.

> 注意
>
> 
>
> The **debmake** command only provides decent 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, focus on understanding the overall process rather than worrying about the details.

If you are familiar with Debian packaging, you'll notice that **debmake** is similar to the **dh_make** command. This is because **debmake** is designed to replace the functionality historically provided by **dh_make**. 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 (see 「項 16.2」, 「第 15 章」, and 「第 16 章」)
- 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 Debian source format「**3.0 (quilt)**」, unless the **-n** option is explicitly specified.

1Before **dh_make**, the **deb-make** command was popular. The current **debmake** package starts its version from **4.0** to avoid version conflicts with the obsolete **debmake** package, which provided the 「**deb-make**」command.

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

ティップ



Ensure that you properly quote the arguments of the **-b**, **-f**, and **-w** options to protect them from shell interference.

ティップ



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

ティップ



The detailed log of all the package build examples in this document can be obtained by following the instructions in 「項 14.14」.

Chapter 3

Prerequisites

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

3.1 People around Debian

Debian 界隈では異なる役割で色々なタイプの人が交流しています:

- **upstream author (**アップストリームの作者**)**: 元のプログラムを作った人です。
- **upstream maintainer (**アップストリームのメンテナー**)**: 現在プログラムをメンテナンスしている人です。
- **maintainer (**メンテナー**)**: プログラムの Debian パッケージを作成している人です。
- **sponsor (**スポンサー**)**: メンテナーのパッケージを、(内容をチェックした後で) 公式 Debian パッケージアーカイブにアップロードするのを手伝う人です。
- **mentor (**指導役**)**: 新米メンテナーをパッケージを作成等で手助けする人です。
- **Debian Developer (Debian** デベロッパー**)** (DD): 公式 Debian パッケージアーカイブへの全面的アップロード権限を持っている Debian プロジェクトのメンバーです。
- **Debian Maintainer (Debian** メンテナー**)** (DM): 公式 Debian パッケージアーカイブへの限定的アップロード権限持っている人です。

Please note that you can't become an official **Debian Developer** (DD) overnight, as it requires more than just technical skills. Don't be discouraged by this. If your work 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 don't need to create new packages to become an official Debian Developer. Contributing to existing packages can also provide a path to becoming an official Debian Developer. There are many packages waiting for good maintainers (see 「"` 項 3.8」`").

3.2 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)

3.3 Debian における社会ダイナミクス

Debian の社会ダイナミクスを理解して、Debian との交流にあなた自身を準備しましょう。

- 我々は全員ボランティアです。
  - 他者に仕事を押し付けられません。
  - 自発的に行動すべきです。
- 友好的な協力が推進力です。
  - あなたの寄与が他者に過大な負荷をかけることがあってはいけません。
  - あなたの寄与は他人に評価されて初めて価値があります。
- Debian は教師から自動的に注意を払ってもらう学校ではありません。
  - 多様な案件を独学できる能力を持つべきです。
  - 他ボランティアに注意を払ってもらうことは貴重なリソースです。
- Debian は常に改良されています。
  - あなたには高品質パッケージを作成することが期待されています。
  - あなたは変化に適応していくべきです。

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)

3.4 Technical reminders

Here are some technical reminders to help other maintainers work on your package easily and effectively, maximizing 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.

注意



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

It is unwise to run your base system under the **unstable** suite, even for development purposes.

- Creation and verification of binary **deb** packages should use a minimal **unstable** chroot as described in 「項 4.7」.

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

- Basic interactive package development activities should use an **unstable** chroot as described in 「項 4.8」.

注意



Advanced package development activities, such as testing full Desktop systems, network daemons, and system installer packages, should use the **unstable** suite running under 「virtualization」.

3.5 Debian documentation

Please make yourself ready to read the pertinent part of the latest Debian documentation to generate perfect Debian packages:

- 「Debian Policy Manual」
  - The official 「must follow」 rules (https://www.debian.org/doc/devel-manuals#policy)
- 「Debian Developer's Reference」
  - The official 「best practice」 document (https://www.debian.org/doc/devel-manuals#devref)
- 「Debian メンテナー用ガイド」— 本文書
  - A 「tutorial reference」 document (https://www.debian.org/doc/devel-manuals#debmake-doc)

All these documents are published on https://www.debian.org using the **unstable** suite versions of corresponding Debian packages. If you wish to have local access to all these documents from your base system, please consider using techniques such as 「apt-pinning」 and 「chroot」.

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 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/
- 「Debian New Maintainers' Guide」(predecessor of this tutorial, deprecated)
  - https://www.debian.org/doc/devel-manuals#maint-guide
  - https://packages.qa.debian.org/m/maint-guide.html

ティップ



When reading these, you may consider using the **debmake** command in place of the **dh_make** command.

3.6 Help resources

Before deciding to ask your question in a public forum, please do your part by reading the relevant 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」.

You can find your desired information 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.

Creating a small test package is a good way to learn the details of 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」

More experienced Debian developers will gladly help you if you ask properly after making the required efforts.

注意



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

3.7 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.

ティップ



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

ティップ



The **how-can-i-help** package can show opportunities for contributing to Debian based on packages installed locally.

3.8 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
  - Debian wiki 「Teams」 page
- 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」.

3.9 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.

2This 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.

  - Send patches to the Debian packaging helper scripts such as **devscripts**, **sbuild**, **schroot**, 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 4

Tool Setups

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

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

It is a good idea to install and set up all of the popular set of packages mentioned in this chapter. These enable us to share the common baseline working environment, although these are not necessarily absolute requirements.

Please also consider to install the tools mentioned in the 「Overview of Debian Maintainer Tools」 in the 「Debian Developer's Reference」, as needed.

> 注意
>
> 
>
> 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.

4.1 Email setup

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

Let's set these environment variables by adding the following lines to **\~/.bashrc** 1.

Add to the ~/.bashrc file

```
DEBEMAIL="osamu@debian.org"
DEBFULLNAME="Osamu Aoki"
export DEBEMAIL DEBFULLNAME
```

> 注意
>
> 
>
> The above is for the author of this manual. The configuration and operation examples presented in this manual use these email address and name settings. You must use your email address and name for your system.

1This 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**.

4.2 mc setup

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
if [ -f /usr/lib/mc/mc.sh ]; then
  . /usr/lib/mc/mc.sh
fi
```

4.3 git setup

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 "Osamu Aoki"
[~] $ git config --global user.email osamu@debian.org
```

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
```

ティップ



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

4.4 quilt setup

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"
. /usr/share/bash-completion/completions/quilt
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 $QUILT_PATCHES
    QUILT_PATCHES="debian/patches"
    QUILT_PATCH_OPTS="--reject-format=unified"
    QUILT_DIFF_ARGS="-p ab --no-timestamps --no-index --color=auto"
    QUILT_REFRESH_ARGS="-p ab --no-timestamps --no-index"
    QUILT_COLORS="diff_hdr=1;32:diff_add=1;34:diff_rem=1;31:diff_hunk=1;33:"
    QUILT_COLORS="${QUILT_COLORS}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 (**quilt.html**)」on how to use the **quilt** command.

See 「項 5.9」for example usages.

Note that 「gbp pq」is able to consume existing **debian/patches**, automate updating and modifying the patches, and export them back into **debian/patches**, all without using quilt nor the need to learn or configure quilt.

4.5 openPGP setup

The **gpg**(1) command, included in the **gnupg** package, can be used to generate a new openPGP key for the Debian package activity as follows.

```
[~] $ gpg --generate-key
gpg (GnuPG) 2.4.7; Copyright (C) 2024 g10 Code GmbH
... [snip] ...
public and secret key created and signed.

pub   ed25519 2026-02-10 [SC] [expires: 2029-02-09]
      ABCDEF0123456789ABCDEF0123456789ABCDEF01
uid                      Osamu Aoki  <osamu@debian.org>
sub   cv25519 2026-02-10 [E] [expires: 2029-02-09]
```

This is an openPGP key based on the Elliptic-curve cryptography (ECC).

注意



If you already have an openPGP key based on the RSA cryptosystem signed by many others, you can use it.

You can edit and update this openPGP key with:

```
[~] $ gpg --edit-key ABCDEF0123456789ABCDEF0123456789ABCDEF01
... [snip] ...
gpg> save
```

For example, the resulting openPGP key may look like:

```
[~] $ gpg --list-keys ABCDEF0123456789ABCDEF0123456789ABCDEF01
pub   ed25519 2026-02-10 [SC]
      ABCDEF0123456789ABCDEF0123456789ABCDEF01
uid           [ultimate] Osamu Aoki <osamu@debian.org>
sub   cv25519 2026-02-10 [E] [expires: 2030-02-10]
```

注意



The term 「GPG key」found in many Debian documents can be considered as a synonym of 「openPGP key」.

For more on OpenPGP key, see:

- 「Creating a new GPG key」
- 「Debian wiki: Keysigning」
- 「Debian wiki: Using OpenPGP subkeys in Debian development」
- 「Debian wiki: FixKeyWithSha1」 — updating old OpenPGP keys

4.6 devscripts setup

The **debsign** command, included in the **devscripts** package, is used to sign the Debian package with your private openPGP 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_openPGP_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 「第 8 章」).

4.7 sbuild setup

The **sbuild** package provides a clean room (「chroot」) build environment. It offers this efficiently with the help of **schroot** using the bind-mount feature of the modern Linux kernel.

Since it is the same build environment as the Debian's buildd infrastructure, it is always up to date and comes full of useful features.

It can be customized to offer following features:

- The **schroot** package to boost the chroot creation speed.
- The **lintian** package to find bugs in the package.
- The **piuparts** package to find bugs in the package.
- The **autopkgtest** package to find bugs in the package.
- 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)

Let's set up **sbuild** environment 2:

```
[~] $ sudo apt install sbuild piuparts autopkgtest lintian
[~] $ sudo apt install sbuild-debian-developer-setup
[~] $ sudo sbuild-debian-developer-setup -s unstable
```

Let's update your group membership by rebooting the entire system 3:

```
[~] $ sudo shutdown -r now
```

Let's verify that your group membership includes **sbuild**:

```
[~] $ id
uid=1000(<yourname>) gid=1000(<yourname>) groups=...,132(sbuild)
```

2Be careful since some older HOWTOs may use different chroot setups.

3Under some modern GUI Desktop environment, 「logout and login using GUI」 may not update your system environment such as group membership.

Let's create the configuration file **~/.local/sbuild/config.pl** in line with recent Debian practice of 「source-only-upload」 as:

```
[~] $ cat >~/.local/sbuild/config.pl << 'EOF'
################################################################################
# PACKAGE BUILD RELATED (source-only-upload as default)
################################################################################
# -d
$distribution = 'unstable';
# -A
$build_arch_all = 1;
# -s
$build_source = 1;
# --source-only-changes
$source_only_changes = 1;
# -v
$verbose = 1;

################################################################################
# POST-BUILD RELATED (turn off functionality by setting variables to 0)
################################################################################
$run_lintian = 1;
$lintian_opts = ['-i', '-I'];
$run_piuparts = 1;
$piuparts_opts = ['--schroot', 'unstable-amd64-sbuild'];
$run_autopkgtest = 1;
$autopkgtest_root_args = '';
$autopkgtest_opts = [ '--', 'schroot', '%r-%a-sbuild' ];

################################################################################
# PERL MAGIC
################################################################################
1;
EOF
```

注意



There are some exceptional cases such as NEW uploads, uploads with NEW binary packages, and security uploads where you can't do source-only-upload but are required to upload with binary packages. The above configuration needs to be adjusted for those exceptional cases.

ティップ



You may need to add 「**$chroot_mode = "schroot";**」to **~/.local/sbuild/config.pl** for **piuparts** if it doesn't work well under **unshare**. See Debian bug: #1125784 and #1126127.

Following document assumes that **sbuild** is configured this way.

Edit this to your needs. Post-build tests can be turned on and off by assigning 1 or 0 to the corresponding variables,

警告



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

注意



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

ティップ



Many **sbuild** related hints are available at「項 9.7」and「https://wiki.debian.org/-sbuild」.

4.8 Persistent chroot setup

注意



Use of independent copied chroot filesystem prevents contaminating the source chroot used by **sbuild**.

For building new experimental packages or for debugging buggy packages, let's setup dedicated persistent chroot 「**source:unstable-amd64-desktop**」by:

```
[~] $ sudo cp -a /srv/chroot/unstable-amd64-sbuild /srv/chroot/unstable-amd64- ↩
    desktop
[~] $ sudo tee /etc/schroot/chroot.d/unstable-amd64-desktop-XXXXXX << EOF
[unstable-amd64-desktop]
description=Debian sid/amd64 persistent chroot
groups=root,sbuild
root-groups=root,sbuild
profile=desktop
type=directory
directory=/srv/chroot/unstable-amd64-desktop
union-type=overlay
EOF
```

Here, **desktop** profile is used instead of **sbuild** profile. Please make sure to adjust **/etc/schroot/desktop/fstab** to make package source accessible from inside of the chroot.

You can log into this chroot 「**source:unstable-amd64-desktop**」by:

```
[~] $ sudo schroot -c source:unstable-amd64-desktop
```

4.9 gbp setup

The **git-buildpackage** package offers the **gbp**(1) command. Its user configuration file is **~/.gbp.conf**.

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

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

4.10 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:

- Specialized HTTP caching proxy using the **apt-cacher-ng** package.
- Generic HTTP caching proxy (**squid** package) configured by **squid-deb-proxy** package

In order to use this HTTP proxy without manual configuration adjustment, it's a good idea to install either **auto-apt-proxy** or **squid-deb-proxy-client** package to everywhere.

4.11 Private Debian repository

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

4.12 Virtual machines

For testing GUI application, it is a good idea to have virtual machines. Install **virt-manager** and **qemu-kvm** packages.

Use of chroot and virtual machines allows us not to update the whole host PC to the latest **unstable** suite.

4.13 Local network with virtual machines

In order to access virtual machines easily over the local network, setting up multicast DNS service discovery infrastructure by installing **avahi-utils** is a good idea.

For all running virtual machines and the host PC, we can use each host name appended with **.local** for SSH to access each other.

Chapter 5

Simple packaging

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」).

5.1 Packaging tarball

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.xz**.

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

Basics for the install from the upstream tarball

```
[base_dir] $ tar --xz -xmf debhello-0.0.tar.xz
[base_dir] $ cd debhello-0.0
[debhello-0.0] $ make
[debhello-0.0] $ 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**.

> 注意
>
> 
>
> Examples of creating a Debian package from other complicated build systems are described in 「第 14 章」.

5.2 Big picture

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

- The maintainer obtains the upstream tarball **debhello-0.0.tar.xz** 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.xz** symlink is created pointing to the **debhello-0.0.tar.xz** 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

```
[base_dir] $ tar --xz -xmf debhello-0.0.tar.xz
[base_dir] $ cd debhello-0.0
[debhello-0.0] $ debmake
I: debmake (version: 5.1.5)
I: Copyright © 2014-2026 Osamu Aoki <osamu@debian.org>
   ...
[debhello-0.0] $ ... manual customization of debian/* files
[debhello-0.0] $ debuild
   ...
```

ティップ



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

ティップ



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.

5.3 What is debmake?

注意



Actual packaging activities are often performed manually without using **debmake** while referencing only existing similar packages and 「Debian Policy Manual」.

The **debmake** command is the helper script for the Debian packaging. (「第 15 章」)

- It creates good template files for the Debian packages.
- 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 provides short extracted license texts as **debian/copyright** using **licensecheck** to help license review.

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

In retrospective, I created **debmake** to simplify this documentation. I consider **debmake** to be more-or-less a demonstration session generator for tutorial purpose.

The **debmake** command isn't the only helper script to make a Debian package. If you are interested alternative packaging helper tools, please see:

- Debian wiki: 「AutomaticPackagingTools」 — Extensive comparison of packaging helper scripts
- Debian wiki: 「CopyrightReviewTools」 — Extensive comparison of copyright review helper scripts

5.4 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 「項 4.6」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 **sbuild** command is a wrapper script to build the Debian binary package under the proper chroot environment with the proper environment variables.

注意



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

5.5 Step 1: Get the upstream source

Let's get the upstream source.

Download debhello-0.0.tar.xz

```
[base_dir] $ wget http://www.example.org/download/debhello-0.0.tar.xz
 ...
[base_dir] $ tar --xz -xmf debhello-0.0.tar.xz
[base_dir] $ tree
.
+-- debhello-0.0
|   +-- Makefile
|   +-- README.md
|   +-- src
|       +-- hello.c
```

```
+-- debhello-0.0.tar.xz

3 directories, 4 files
```

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

hello.c

```
[base_dir] $ 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

```
[base_dir] $ 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
```

注意



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

5.6 Step 2: Generate template files with debmake

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

The output from the debmake command with -x1 option

```
[base_dir] $ cd debhello-0.0
[debhello-0.0] $ debmake -x1
I: debmake (version: 5.1.5)
I: Copyright © 2014-2026 Osamu Aoki <osamu@debian.org>
I: [debhello-0.0] $ cd ..
I: Non-native Debian package pkg="debhello", ver="0.0", rev="1" method="dir_d...
I: already in the package-version form: "debhello-0.0"
I: [base_dir] $ ln -sf 'debhello-0.0.tar.xz' 'debhello_0.0.orig.tar.xz'
I: [base_dir] $ cd debhello-0.0
I: parsing option -b ""
I: binary package=debhello Type=bin / Arch=any M-A=foreign
I: build_type = make
I: ext_type = c                       1 files
I: ext_type = md                      1 files
I: creating debian/* files with "-x 1" option
I: [debhello-0.0] $ licensecheck --recursive --copyright --deb-machine  . > d...
I: creating debian/copyright by licensecheck.
I: creating debian/control from control.py
I: creating debian/control by control.py
I: creating debian/changelog from extra0_changelog
I: creating debian/rules from extra0_rules
I: creating debian/source/format from extra0source_format
I: creating debian/README.Debian from extra1_README.Debian
I: creating debian/README.source from extra1_README.source
I: creating debian/clean from extra1_clean
I: creating debian/dirs from extra1_dirs
I: creating debian/docs from extra1_docs
I: creating debian/examples from extra1_examples
I: creating debian/gbp.conf from extra1_gbp.conf
I: creating debian/links from extra1_links
I: creating debian/manpages from extra1_manpages
I: creating debian/salsa-ci.yml from extra1_salsa-ci.yml
I: creating debian/watch from extra1nn_watch
I: creating debian/tests/control from extra1tests_control
I: creating debian/upstream/metadata from extra1upstream_metadata
I: creating debian/patches/series from extra1patches_series
I: creating debian/install from extra1bin_install
I: [debhello-0.0] $ wrap-and-sort -ast
I: debian/* may have a blank line at the top.
```

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** (without maintainer script supports for simplicity)

注意



Here, the **debmake** command is invoked with the **-x1** option to keep this tutorial simple. Use of default **-x2** or more extensive **-x3** option is highly recommended.

Let's inspect generated template files.

The source tree after the basic debmake execution.

```
[debhello-0.0] $ cd ..
[base_dir] $ tree
.
+-- debhello-0.0
|   +-- Makefile
|   +-- README.md
|   +-- debian
|   |   +-- README.Debian
|   |   +-- README.source
|   |   +-- changelog
|   |   +-- clean
|   |   +-- control
|   |   +-- copyright
|   |   +-- dirs
|   |   +-- docs
|   |   +-- examples
|   |   +-- gbp.conf
|   |   +-- install
|   |   +-- links
|   |   +-- manpages
|   |   +-- patches
|   |   |   +-- series
|   |   +-- rules
|   |   +-- salsa-ci.yml
|   |   +-- source
|   |   |   +-- format
|   |   +-- tests
|   |   |   +-- control
|   |   +-- upstream
|   |   |   +-- metadata
|   |   +-- watch
|   +-- src
|       +-- hello.c
+-- debhello-0.0.tar.xz
+-- debhello_0.0.orig.tar.xz -> debhello-0.0.tar.xz

8 directories, 25 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):

```
[base_dir] $ cd debhello-0.0
[debhello-0.0] $ cat debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
# See debhelper(7) (un-comment to enable)
# This is an autogenerated template for debian/rules.
#
# Output every command that modifies files on the build system.
#export DH_VERBOSE = 1
#
# Copy some variable definitions from pkg-info.mk and vendor.mk
# under /usr/share/dpkg/ to here if they are useful.
#
# See FEATURE AREAS/ENVIRONMENT in dpkg-buildflags(1)
# Apply all hardening options
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
# Package maintainers to append CFLAGS
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
# Package maintainers to append LDFLAGS
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,-O1
```

```
#
# With debhelper version 9 or newer, the dh command exports
# all buildflags.  So there is no need to include the
# /usr/share/dpkg/buildflags.mk file here if compat is 9 or newer.
#
# These are rarely used code. (START)
#
# The following include for *.mk magically sets miscellaneous
# variables while honoring existing values of pertinent
# environment variables:
#
# Architecture-related variables such as DEB_TARGET_MULTIARCH:
#include /usr/share/dpkg/architecture.mk
# Vendor-related variables such as DEB_VENDOR:
#include /usr/share/dpkg/vendor.mk
# Package-related variables such as DEB_DISTRIBUTION
#include /usr/share/dpkg/pkg-info.mk
#
# You may alternatively set them susing a simple script such as:
# DEB_VENDOR ?= $(shell dpkg-vendor --query  Vendor)
#
# These are rarely used code. (END)
#

### main packaging script based on post dh7 syntax
%:
        dh $@

# debmake generated override targets
# Use "make prefix=/usr" (override prefix=/usr/local in Makefile)
#override_dh_auto_install:
#       dh_auto_install -- prefix=/usr

# Do not install python .pyc .pyo if they exist
#override_dh_install:
#       dh_install --list-missing -X.pyc -X.pyo

# Multiarch package requires library files to be installed to
# /usr/lib/<triplet>/ .  If the build system does not support
# $(DEB_HOST_MULTIARCH), you may need to override some targets such as
# dh_auto_configure or dh_auto_install to use $(DEB_HOST_MULTIARCH) .
```

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):

```
[debhello-0.0] $ cat debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: "Osamu Aoki" <osamu@debian.org>
Build-Depends:
 debhelper-compat (= 13),
Standards-Version: 4.7.3
Homepage: <insert the upstream URL, if relevant>
Rules-Requires-Root: no
#Vcs-Git: https://salsa.debian.org/debian/debhello.git
#Vcs-Browser: https://salsa.debian.org/debian/<project_site>

Package: debhello
Section: unknown
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.
 .
 ===== This comes from the unmodified template file =====
 .
 Please edit this template file (debian/control) and other package files
 (debian/*) to make them meet all the requirements of the Debian Policy
 before uploading this package to the Debian archive.
 .
 See
  * https://www.debian.org/doc/manuals/developers-reference/best-pkging-pract...
  * https://www.debian.org/doc/manuals/debmake-doc/ch05.en.html#control
 .
 The synopsis description at the top should be about 60 characters and
 written as a phrase.  No extra capital letters or a final period.  No
 articles b''—b'' "a", "an", or "the".
 .
 The package description for general-purpose applications should be
 written for a less technical user.  This means that we should avoid
 jargon.  GNOME or KDE is fine but GTK+ is probably not.
 .
 Use the canonical forms of words:
  * Use X Window System, X11, or X; not X Windows, X-Windows, or X Window.
  * Use GTK+, not GTK or gtk.
  * Use GNOME, not Gnome.
  * Use PostScript, not Postscript or postscript.
```

警告



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 「第 6 章」.

注意



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 using the **licensecheck** command.

debian/copyright (template file):

```
[debhello-0.0] $ cat debian/copyright
Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: FIXME
Upstream-Contact: FIXME
Source: FIXME
Disclaimer: Autogenerated by licensecheck
```

```
Files: ./Makefile
 ./README.md
 ./src/hello.c
Copyright: NONE
License: UNKNOWN
 FIXME
```

5.7 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):

```
[base_dir] $ cd debhello-0.0
[debhello-0.0] $ vim debian/rules
 ... hack, hack, hack, ...
[debhello-0.0] $ 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 $@

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):

```
[debhello-0.0] $ vim debian/control
 ... hack, hack, hack, ...
[debhello-0.0] $ cat debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends:
 debhelper-compat (= 13),
Standards-Version: 4.7.3
Homepage: https://salsa.debian.org/debian/debmake-doc
```

1This 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.

2This 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.

```
Rules-Requires-Root: no

Package: debhello
Architecture: any
Multi-Arch: foreign
Depends:
 ${misc:Depends},
 ${shlibs:Depends},
Description: Simple packaging example for debmake
 This Debian binary package is an example package.
 (This is an example only)
```

debian/copyright (maintainer version):

```
[debhello-0.0] $ vim debian/copyright
 ... hack, hack, hack, ...
[debhello-0.0] $ cat debian/copyright
Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: debhello
Upstream-Contact: Osamu Aoki <osamu@debian.org>
Source: https://salsa.debian.org/debian/debmake-doc

Files:     *
Copyright: 2015-2021 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.
```

Let's remove unused template files and edit remaining template files:

- **debian/README.source**
- **debian/patches/series** (No upstream patch)
- **clean**
- **dirs**
- **install**
- **links**

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

```
[debhello-0.0] $ rm -f debian/clean debian/dirs debian/install debian/links
[debhello-0.0] $ rm -f debian/README.source debian/source/*.ex
[debhello-0.0] $ rm -rf debian/patches
[debhello-0.0] $ tree -F debian
debian/
+-- README.Debian
+-- changelog
```

```
+-- control
+-- copyright
+-- docs
+-- examples
+-- gbp.conf
+-- manpages
+-- rules*
+-- salsa-ci.yml
+-- source/
|   +-- format
+-- tests/
|   +-- control
+-- upstream/
|   +-- metadata
+-- watch

4 directories, 14 files
```

ティップ



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

5.8 Step 4: Building package with debuild

You can create a non-native Debian package using the **debuild** command or its equivalents (see 「項 5.4」) in this source tree. The command output is very verbose and explains what it does as follows.

Building package with debuild

```
[base_dir] $ cd debhello-0.0
[debhello-0.0] $ debuild
 dpkg-buildpackage -us -uc -ui -i
dpkg-buildpackage: info: source package debhello
dpkg-buildpackage: info: source version 0.0-1
dpkg-buildpackage: info: source distribution unstable
dpkg-buildpackage: info: source changed by Osamu Aoki <osamu@debian.org>
 dpkg-source -i --before-build .
dpkg-buildpackage: info: host architecture amd64
 debian/rules clean
dh clean
   dh_auto_clean
        make -j12 distclean
 ...
 debian/rules binary
dh binary
   dh_update_autotools_config
   dh_autoreconf
   dh_auto_configure
   dh_auto_build
        make -j12 INSTALL="install --strip-program=true"
make[1]: Entering directory '/path/to/base_dir/debhello-0.0'
# CFLAGS=-g -O2 -Werror=implicit-function-declaration
 ...
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 「第 14 章」). Let's move on for now.

Let's inspect the result.

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

```
[debhello-0.0] $ cd ..
[base_dir] $ tree -FL 1
./
+-- debhello-0.0/
+-- debhello-0.0.tar.xz
+-- 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.xz -> debhello-0.0.tar.xz

2 directories, 9 files
```

You see all the generated files.

- The **debhello_0.0.orig.tar.xz** 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 「項 10.21」.
- 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:

```
[base_dir] $ tar --xz -tf debhello-0.0.tar.xz
debhello-0.0/
debhello-0.0/src/
debhello-0.0/src/hello.c
debhello-0.0/Makefile
debhello-0.0/README.md
[base_dir] $ tar --xz -tf debhello_0.0-1.debian.tar.xz
debian/
debian/README.Debian
debian/changelog
debian/control
debian/copyright
debian/docs
debian/examples
debian/gbp.conf
debian/manpages
debian/rules
debian/salsa-ci.yml
debian/source/
debian/source/format
debian/tests/
debian/tests/control
debian/upstream/
debian/upstream/metadata
debian/watch
```

The **debhello_0.0-1_amd64.deb** contains the binary files to be installed to the target system.

The **debhello-dbgsym_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:

```
[base_dir] $ 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/46/
-rw-r--r-- root/root ...  ./usr/lib/debug/.build-id/46/b45380346979015bc7a113...
drwxr-xr-x root/root ...  ./usr/share/
drwxr-xr-x root/root ...  ./usr/share/doc/
lrwxrwxrwx root/root ...  ./usr/share/doc/debhello-dbgsym -> debhello
[base_dir] $ 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):

```
[debhello-0.0] $ dpkg -f debhello-dbgsym_0.0-1_amd64.deb pre-depends \
           depends recommends conflicts breaks
Depends: debhello (= 0.0-1)
[debhello-0.0] $ dpkg -f debhello_0.0-1_amd64.deb pre-depends \
           depends recommends conflicts breaks
Depends: libc6 (>= 2.34)
```

注意



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

注意



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.

5.9 Step 3 (alternatives): 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 modify files in the upstream source. For example, **Makefile** may

be modified to set the **$(prefix)** value to **/usr**.

注意



The above 「項 5.7」using the **debian/rules** file is the better approach for packaging for this example. But let's continue on with this alternative approaches as a leaning experience.

In the following, let's consider 3 simple variants of this alternative approach to generate **debian/patches/*** files representing modifications to the upstream source in the Debian source format「**3.0 (quilt)**」. These substitute 「項 5.7」in the above step-by-step example:

- 「項 5.10」
- 「項 5.11」
- 「項 5.12」

Please note the **debian/rules** file used for these examples doesn't have the **override_dh_auto_install** target as follows:

debian/rules (alternative maintainer version):

```
[base_dir] $ cd debhello-0.0
[debhello-0.0] $ vim debian/rules
 ... hack, hack, hack, ...
[debhello-0.0] $ 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 $@
```

5.10 Patch by「diff -u」approach

Here, the patch file **000-prefix-usr.patch** is created using the **diff** command.

Patch by diff -u

```
[base_dir] $ cp -a debhello-0.0 debhello-0.0.orig
[debhello-0.0] $ vim debhello-0.0/Makefile
 ... hack, hack, hack, ...
[base_dir] $ diff -Nru debhello-0.0.orig debhello-0.0 >000-prefix-usr.patch
[base_dir] $ cat 000-prefix-usr.patch
diff -Nru debhello-0.0.orig/Makefile debhello-0.0/Makefile
--- debhello-0.0.orig/Makefile  2026-07-22 15:28:48.183844548 +0900
+++ debhello-0.0/Makefile       2026-07-22 15:28:48.271746924 +0900
@@ -1,4 +1,4 @@
-prefix = /usr/local
+prefix = /usr

 all: src/hello

[base_dir] $ rm -rf debhello-0.0
[base_dir] $ mv -f debhello-0.0.orig debhello-0.0
```

Please note that the upstream source tree is restored to the original state after generating a patch file **000-prefix-usr.patch**.

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

000-prefix-usr.patch (DEP-3):

```
[debhello-0.0] $ echo '000-prefix-usr.patch' >debian/patches/series
[debhello-0.0] $ vim ../000-prefix-usr.patch
 ... hack, hack, hack, ...
[debhello-0.0] $ mv -f ../000-prefix-usr.patch debian/patches/000-prefix-usr....
[debhello-0.0] $ 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
```

注意



When generating the Debian source package by **dpkg-source** via **dpkg-buildpackage** in the following step of「項 5.8」, the **dpkg-source** command assumes that no patch was applied to the upstream source, since the **.pc/applied-patches** is missing.

5.11 Patch by dquilt approach

Here, the patch file **000-prefix-usr.patch** is created using the **dquilt** command.

dquilt 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 generated patch is stored in the **debian/patches/** directory.

Patch by dquilt

```
[debhello-0.0] $ dquilt new 000-prefix-usr.patch
Patch debian/patches/000-prefix-usr.patch is now on top
[debhello-0.0] $ dquilt add Makefile
File Makefile added to patch debian/patches/000-prefix-usr.patch
 ... hack, hack, hack, ...
[debhello-0.0] $ head -1 Makefile
prefix = /usr
[debhello-0.0] $ dquilt refresh
Refreshed patch debian/patches/000-prefix-usr.patch
[debhello-0.0] $ dquilt header -e --dep3
 ... edit the DEP-3 patch header with editor
[debhello-0.0] $ tree -a
.
+-- .pc
|   +-- .quilt_patches
|   +-- .quilt_series
|   +-- .version
|   +-- 000-prefix-usr.patch
|   |   +-- .timestamp
|   |   +-- Makefile
|   +-- applied-patches
+-- Makefile
+-- README.md
+-- debian
|   +-- README.Debian
|   +-- README.source
```

```
|   +-- changelog
|   +-- clean
|   +-- control
|   +-- copyright
|   +-- dirs
|   +-- docs
|   +-- examples
|   +-- gbp.conf
|   +-- install
|   +-- links
|   +-- manpages
|   +-- patches
|   |   +-- 000-prefix-usr.patch
|   |   +-- series
|   +-- rules
|   +-- salsa-ci.yml
|   +-- source
|   |   +-- format
|   +-- tests
|   |   +-- control
|   +-- upstream
|   |   +-- metadata
|   +-- watch
+-- src
    +-- hello.c

9 directories, 30 files
[debhello-0.0] $ cat debian/patches/series
000-prefix-usr.patch
[debhello-0.0] $ 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 for the packaging.

注意



When generating the Debian source package by **dpkg-source** via **dpkg-buildpackage** in the following step of「項 5.8」, the **dpkg-source** command assumes that patches were applied to the upstream source, since the **.pc/applied-patches** exists.

The upstream source tree can be restored to the original state for the packaging.

The upstream source tree (restored):

```
[debhello-0.0] $ dquilt pop -a
Removing patch debian/patches/000-prefix-usr.patch
Restoring Makefile

No patches applied
[debhello-0.0] $ head -1 Makefile
prefix = /usr/local
```

```
[debhello-0.0] $ tree -a .pc
.pc
+-- .quilt_patches
+-- .quilt_series
+-- .version

1 directory, 3 files
```

Here, **Makefile** is restored and the **.pc/applied-patches** is missing.

5.12 Patch by「dpkg-source --auto-commit」approach

Here, the patch file isn't created in this step but the source files are setup to create **debian/patches/***
files in the following step of「項 5.8」.

For this, **debmake** must be invoked without **-x1** option to generate normal template files using default **-x2** option, instead.

The output from the debmake command

```
[base_dir] $ cd debhello-0.0
[debhello-0.0] $ debmake
I: debmake (version: 5.1.5)
I: Copyright © 2014-2026 Osamu Aoki <osamu@debian.org>
...
```

Let's edit the upstream source.

Modified Makefile

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

Let's edit **debian/source/options**:

debian/source/options for auto-commit

```
[debhello-0.0] $ mv debian/source/options.ex debian/source/options
[debhello-0.0] $ vim debian/source/options
 ... hack, hack, hack, ...
[debhello-0.0] $ cat debian/source/options
# == Patch applied strategy (merge) ==
#
# The source outside of debian/ directory is modified by maintainer and
# different from the upstream one:
#   * Workflow using dpkg-source commit (commit all to VCS after dpkg-source ...
#       https://www.debian.org/doc/manuals/debmake-doc/ch04.en.html#dpkg-sour...
#   * Workflow described in dgit-maint-merge(7)
#
single-debian-patch
auto-commit
```

Let's edit **debian/source/patch-header**:

debian/source/patch-header for auto-commit

```
[debhello-0.0] $ mv debian/source/patch-header.ex debian/source/patch-header
[debhello-0.0] $ vim debian/source/patch-header
 ... hack, hack, hack, ...
[debhello-0.0] $ cat debian/source/patch-header
Description: debian-changes
Author: Osamu Aoki <osamu@debian.org>
```

Let's remove **debian/patches/*** files and other unused template files.

Remove unused template files

```
[debhello-0.0] $ rm -f debian/clean debian/dirs debian/install debian/links
[debhello-0.0] $ rm -f debian/README.source debian/*.ex debian/source/*.ex
[debhello-0.0] $ rm -rf debian/patches
[debhello-0.0] $ tree debian
debian
+-- README.Debian
+-- changelog
+-- control
+-- copyright
+-- docs
+-- examples
+-- gbp.conf
+-- manpages
+-- rules
+-- salsa-ci.yml
+-- source
|   +-- format
|   +-- options
|   +-- patch-header
+-- tests
|   +-- control
+-- upstream
|   +-- metadata
+-- watch

4 directories, 16 files
```

There are no **debian/patches/*** files at the end of this step.

注意



When generating the Debian source package by **dpkg-source** via **dpkg-buildpackage** in the following step of 「項 5.8」, the **dpkg-source** command uses options specified in **debian/source/options** to auto-commit modification applied to the upstream source as **patches/debian-changes**.

Let's inspect the Debian source package generated after the following 「項 5.8」step and extracting files from **debhello-0.0.debian.tar.xz**.

Inspect debhello-0.0.debian.tar.xz after debuild

```
[base_dir] $ tar --xz -xvf debhello_0.0-1.debian.tar.xz
debian/
debian/README.Debian
debian/changelog
debian/control
debian/copyright
debian/docs
debian/examples
debian/gbp.conf
debian/manpages
debian/patches/
debian/patches/debian-changes
debian/patches/series
debian/rules
debian/salsa-ci.yml
debian/source/
debian/source/format
debian/source/options
debian/source/patch-header
debian/tests/
debian/tests/control
debian/upstream/
```

```
debian/upstream/metadata
debian/watch
```

Let's check generated **debian/patches/*** files.

Inspect debian/patches/* after debuild

```
[base_dir] $ cat debian/patches/series
debian-changes
[base_dir] $ cat debian/patches/debian-changes
Description: debian-changes
Author: Osamu Aoki <osamu@debian.org>

--- debhello-0.0.orig/Makefile
+++ debhello-0.0/Makefile
@@ -1,4 +1,4 @@
-prefix = /usr/local
+prefix = /usr

 all: src/hello
```

The Debian source package **debhello-0.0.debian.tar.xz** is confirmed to be generated properly with **debian/patches/*** files for the Debian modification.

Chapter 6

Basics for packaging

Here, a broad overview is presented without using VCS operations for the basic rules of Debian packaging focusing on the non-native Debian package in the 「**3.0 (quilt)**」 format.

注意



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 Debian source format「**3.0 (quilt)**」is the most normal Debian source package format.

注意



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

6.1 Packaging workflow

The Debian packaging workflow to create a Debian binary package involves generating several specifically named files (see 「項 6.3」) as defined in the 「Debian Policy Manual」. This workflow can be summarized in 10 steps with some over simplification as follows.

1. The upstream tarball is downloaded as the *package-version***.tar.xz** 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.xz**.
   - 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 「項 6.5」)

  debian/control The package configuration file containing the source package name, the source build dependencies, the binary package name, the binary dependencies, etc. (see 「項 6.6」)

  debian/changelog The Debian package history file defining the upstream package version and the Debian revision in its first line (see 「項 6.7」)

  debian/copyright The copyright and license summary (see 「項 6.8」)

  debian/source/format This indicates the desired format to **dpkg-source**(1) (see Debian wiki: 「DebSrc3.0」)

- Optional specification files under the **debian/*** (see 「項 6.14」):
- 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 **sbuild**) 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 Debian source format 「**3.0 (quilt)**」using **dpkg-source**(1)
     - *package_version***.orig.tar.xz** (copy or symlink of *package-version***.tar.xz**)
     - *package_version-revision***.debian.tar.xz** (tarball of **debian/** found in *package-version***/**)
     - *package_version-revision***.dsc**
   - Build the source using 「**debian/rules build**」into **$(DESTDIR)**
     - 「**DESTDIR=debian/***binarypackage***/**」for single binary package [1]
     - 「**DESTDIR=debian/tmp/**」for 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**
     - *package_version-revision_arch*.**buildinfo**
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. Test the goodness of the generated Debian binary package manually by installing it and running its programs.
8. After confirming the goodness, prepare files for the normal source-only upload to the Debian archive.
9. Sign the Debian package file with the **debsign** command using your private openPGP key.
   - Use 「**debsign** *package_version-revision_***source.changes**」(source-only upload situation)
   - Use 「**debsign** *package_version-revision_arch***.changes**」(source+binary upload situation)
10. Upload the set of the Debian package files with the **dput** command to the Debian archive.
    - Use 「**dput** *package_version-revision_***source.changes**」(source-only upload)

1This is the default up to **debhelper** v13. At **debhelper** v14, it warns the default change. After **debhelper** v15, it will change the default to **DESTDIR=debian/tmp/** .

- Use 「**dput** *package_version-revision_arch***.changes**」(source+binary upload)

Test building and confirming of the binary package goodness as above is the moral obligation as a diligent Debian developer but there is no physical barrier for people to skip such operations at this moment for the source-only upload.

For the upstream tarball, the **debmake** command helps up to the step 4 in the above workflow. For the upstream working tree *package/* checked out, e.g., by「**git clone https://github.com/***upstreamname/package***.git**」without any upstream tarball, the **debmake** command invoked in it helps up to step 4, too. The **debmake** command does not overwrite any existing configuration files.

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 (e.g., **amd64**)

The current Debian practice for uploading the normal Debian package is:

- Use the source-only upload if all generated binary packages exist in the Debian **sid** archive. This is usual case.
- Use the source+binary upload if any one of generated packages is missing in the Debian **sid** archive. (This involves manually handled NEW process by the archive management team.)

See also 「Source-only uploads」.

ティップ



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

ティップ



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

6.2 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「Debian 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 「Debian 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.

注意



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.)

注意



debhelper(1) changes its behavior with time. Please make sure to read **debhelper-compat-upgrade-checklist**(7) to understand the situation.

6.3 パッケージ名とバージョン

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

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」 .

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. 2

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
```

2For 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.

バージョンの比較ルールは以下のようにまとめられます:

- 文字列は頭から尾へと比較されます。
- 英文字は数字よりも大きいです。
- 数字は整数として比較されます。
- 英文字は ASCII コード順で比較されます。

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.xz** as the pre-release of **hello-0.9.12.tar.xz**. You can ensure the Debian package upgrade to work properly by renaming the upstream source to **hello-0.9.12~rc99.tar.xz**.

6.4 ネイティブ Debian パッケージ

The non-native Debian package in the Debian source format 「**3.0 (quilt)**」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.

注意



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

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

ティップ



There is no need to create the tarball in advance if the native Debian package format is used. The **debian/source/format** file should have 「**3.0 (native)**」in it as described in **dpkg-source**(1) and The **debian/source/format** file should have the version without the Debian revision (**1.0** instead of **1.0-1**). Then, the tarball containing is generated when 「**dpkg-source -b**」is invoked in the source tree.

6.5 debian/rules file

The **debian/rules** file is the executable script which re-targets the upstream build system 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 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 $@
```

The **dh** command functions as the sequencer to call all required 「**dh** *target*」commands at the right moment. [4]

- **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

Here, **$(DESTDIR)** path depends on the build type.

- 「**DESTDIR=debian/***binarypackage***/**」for single binary package [5]
- 「**DESTDIR=debian/tmp/**」for multi binary package

See 「項 9.2」and 「項 9.3」for customization.

ティップ



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.

6.6 debian/control file

The **debian/control** file consists of blocks of metadata separated by blank lines. Each block of metadata defines the following, in this order:

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

3The **debmake** command generates a bit more complicated **debian/rules** file. But this is the core part.

4This simplicity is available since version 7 of the **debhelper** package. This guide assumes the use of **debhelper** version 14 or newer.

5This is the default up to **debhelper** v13. At **debhelper** v14, it warns the default change. After **debhelper** v15, it will change the default to **DESTDIR=debian/tmp/** .

See 「Chapter 5 - Control files and their fields」 of the "Debian Policy Manual" for the definition of each metadata field.

注意



The **debmake** command sets the **debian/control** file with 「**Build-Depends: debhelper-compat (= 14)**」 to set the **debhelper** compatibility level.

ティップ



If an existing package has a **debhelper** compatibility level lower than 14, it's probably time to update its packaging.

6.7 debian/changelog file

The **debian/changelog** file records the Debian package history.

- Edit this file using the **debchange** command (alias **dch**).
- The first line defines the upstream package version and the Debian revision.
- Document changes in a specific, formal, and concise style.
  - If Debian maintainer modification fixes reported bugs, add 「**Closes:** #<*bug_number*>」 to close those bugs.
- Even if you're uploading your package yourself, you must document all non-trivial user-visible changes, such as:
  - Security-related bug fixes.
  - User interface changes.
- If you're asking a sponsor to upload it, document changes more comprehensively, including all packaging-related ones, to help with package review.
  - The sponsor shouldn't have to guess your reasoning behind package changes.
  - Remember that the sponsor's time is valuable.

After finishing your packaging and verifying its quality, execute the "**dch -r**" command and save the finalized **debian/changelog** file with the suite normally set to **unstable**. [6] If you're packaging for backports, security updates, LTS, etc., use the appropriate distribution names instead.

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 uploads to the Debian archive.

ティップ



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

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

ティップ



Use a **debian/changelog** entry with a version string like **1.0.1-1~rc1** when experimenting. Later, consolidate such **changelog** entries into a single entry for the official package.

The **debian/changelog** file 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.

6.8 debian/copyright file

Debian takes copyright and license matters very seriously. The ”Debian Policy Manual” requires a summary of these in the **debian/copyright** file of the package.

- 「12.5. Copyright information」
- 「2.3. Copyright considerations」
- 「License information」

The **debmake** command creates the initial **debian/copyright** template file using the **licensecheck**(1) command.

6.9 debian/patches/* files

As demonstrated in 「項 5.9」, the **debian/patches/** directory holds

- *patch-file-name***.patch** files providing **-p1** patches and
- the **series** file which defines how these patches are applied.

See how these files are used in:

- 「項 13.6」to build the Debian source package
- 「項 13.7」to extract source files from the Debian source package

注意



Header texts of these patches should conform to 「DEP-3」.

注意



If you want to use VCS tools such as **git**, **gbp** and **dgit** to create and manage these patches after learning basics here, please refer to later in 「第 11 章」.

6.10 debian/source/include-binaries file

The 「**dpkg-source --commit**」command functions like **dquilt** but has one advantage over the **dquilt** command. The **dquilt** command can't handle modified binary files since they are not representable in a diff. Also, adding binary files under the **debian/** directory is normally rejected by **dpkg-source**. By listing these binary files in **debian/source/include-binaries**, the maintainer can include these binary files to the Debian source package generated by **dpkg-source**.

6.11 debian/watch file

注意



This file is for use by the Debian non-native package.

The **uscan**(1) command downloads the latest upstream version using the **debian/watch** file. E.g.:
Basic debian/watch file:

```
version=4
https://ftp.gnu.org/gnu/hello/ @PACKAGE@@ANY_VERSION@@ARCHIVE_EXT@
```

The **uscan** command may verify the authenticity of the upstream tarball with optional configuration (see 「項 6.12」).
See **uscan**(1), 「項 9.4」, 「項 8.1」, and 「項 11.7」for more.

6.12 debian/upstream/signing-key.asc file

Some packages are signed by a openPGP key and their authenticity can be verified using their public openPGP 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.xz** (upstream source)
- **hello-***version***.tar.xz.sig** (detached signature)

Let's pick the latest version set.
Download the upstream tarball and its signature.

```
[base_dir] $ wget https://ftp.gnu.org/gnu/hello/hello-2.9.tar.xz
 ...
[base_dir] $ wget https://ftp.gnu.org/gnu/hello/hello-2.9.tar.xz.sig
 ...
[base_dir] $ gpg --verify hello-2.9.tar.xz.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 openPGP 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.
Download public openPGP key for the upstream

```
[base_dir] $ 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
[base_dir] $ gpg --verify hello-2.9.tar.xz.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
```

ティップ



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/signi key.asc** file.

Set public openPGP key to debian/upstream/signing-key.asc

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

With the above **debian/upstream/signing-key.asc** file and the following **debian/watch** file, the **uscan** command can verify the authenticity of the upstream tarball after its download. E.g.:

Improved debian/watch file with openPGP support:

```
version=4
opts="pgpsigurlmangle=s/$/.sig/" \
https://ftp.gnu.org/gnu/hello/ @PACKAGE@@ANY_VERSION@@ARCHIVE_EXT@
```

6.13 debian/salsa-ci.yml file

Install Salsa CI configuration file. See 「項 11.3」.

6.14 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.

ティップ



Even an upstream source without its build system can be packaged just by using these files. See 「項 14.2」as an example.

The alphabetical list of notable optional **debian/***binarypackage***.*** configuration files listed below provides very powerful means to set the installation path of files. Please note:

- The 「**-x[01234]**」superscript notation that appears in the following list indicates the minimum value for the **debmake -x** option that generates the associated template file. See 「項 16.4」or **debmake**(1) for details.
- 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.
- Some configuration template files generated by the **debmake** command with an extra **.ex** suffix need to be activated by removing that suffix.
- The **debmake -B** command adds template files with an extra **.ex** suffix for all existing template files without **.ex** and they 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.

binarypackage*.bug-control** **-x2** installed as **usr/share/bug/binarypackage***/control** in *binarypackage*. See 「項 9.11」.

binarypackage*.bug-presubj** **-x2** installed as **usr/share/bug/binarypackage***/presubj** in *binarypackage*. See 「項 9.11」.

binarypackage*.bug-script** **-x2** installed as **usr/share/bug/binarypackage* or **usr/share/bug/***binarypackage***/scr** in *binarypackage*. See 「項 9.11」.

***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 **-x1** 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 **-x4** Set the **debhelper** compatibility level. (deprecated)

Use「**Build-Depends: debhelper-compat (= 14)**」in **debian/control** to specify the compatibility level and remove **debian/compat**.

See 「**COMPATIBILITY LEVELS**」in **debhelper**(7).

***binarypackage*.conffiles** **-x3** This optional file is installed into the **DEBIAN** directory within the binary package while supplementing it with all the conffiles auto-detected by **debhelper**.

This file is primarily useful for using ”special” entries such as the remove-on-upgrade feature from **dpkg**(1).

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 「項 10.22」.

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** -x1 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** -x1 Installed as the **doc-base** control file in *binarypackage*.

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

***binarypackage*.docs** -x1 List documentation files to be installed in *binarypackage*.

See **dh_installdocs**(1).

binarypackage*.emacsen-compat** Installed into **usr/lib/emacsen-common/packages/compat/binarypackag* in *binarypackage*.

See **dh_installemacsen**(1).

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

See **dh_installemacsen**(1).

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

See **dh_installemacsen**(1).

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

See **dh_installemacsen**(1).

binarypackage*.examples** -x1 List example files or directories to be installed into **usr/share/doc/binarypackage* in *binarypackage*.

See **dh_installexamples**(1).

gbp.conf -x1 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** -x1 List info files to be installed in *binarypackage*.

See **dh_installinfo**(1).

binarypackage*.init** -x4 Installed into **etc/init.d/binarypackage* in *binarypackage*. (deprecated)

See **dh_installinit**(1).

***binarypackage*.install** -x1 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).

***binarypackage*.links** -x1 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** -x2 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」.

binarypackage*.maintscript** -x2 If this optional file exists, **debhelper** uses this as the template to generate **DEBIAN/binarypackage***.{pre,post}{inst,rm}** files within the binary package while adding 「**-- "$@"**」 to the **dpkg-maintscript-helper**(1) command.

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

manpage.* -x2 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** -x3 or **manpage.1** -x3 as the starting point.

***binarypackage*.manpages** -x1 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.

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

See **dpkg-source**(1),「項 4.4」and「項 5.9」.

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

***binarypackage*.preinst** -x3**, *binarypackage*.postinst** -x3**, *binarypackage*.prerm** -x3**, *binarypackage*.postrm** -

If these optional files exist, the corresponding files are installed into the **DEBIAN** directory within the binary package after enriched by **debhelper**. Otherwise, these files in the **DEBIAN** directory within the binary package is generated by **debhelper**.

Whenever possible, simpler *binarypackage***.maintscript** should be used instead.

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**.

This file provides the information specific to the Debian package.

See **dh_installdocs**(1).

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

If running「**dpkg-source -x**」on a source package doesn't produce the source of the package, ready for editing, and allow one to make changes and run **dpkg-buildpackage** to produce a modified package without taking any additional steps, creating this file is recommended.

See「Debian policy manual section 4.14」.

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 (popular)
- Use「**3.0 (native)**」to make this native package

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

source/lintian-overrides -x2 This file is not installed, but is scanned by the **lintian** command to provide overrides for the source package.

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

source/local-options and source/local-patch-header -x4

> 注意
>
> 
>
> These files are not compatible with the **dgit** workflow. See「項 11.14」.

There is no reason to use these with the current version of **dpkg-source**(1).

source/options [-x2] The **dpkg-source** command uses this content as its options. This is typically used with 「項 11.13」and options may be:

- **auto-commit**
- **single-debian-patch**

This is included in the generated source package.

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

source/patch-header [-x2] Free form text that is put on top of the automatic patch generated.

This is included in the generated source package and is meant to be committed to the "` 項 11.13`.".

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

***binarypackage*.symbols** [-x1] The symbols files, if present, are passed to the **dpkg-gensymbols** command to be processed and installed.

See **dh_makeshlibs**(1) and 「項 10.16」..

***binarypackage*.templates** This is the **debconf templates** file used for asking any questions necessary to configure the package. See 「項 10.22」.

tests/control [-x1] This is the RFC822-style test meta data file defined in DEP-8. See **autopkgtest**(1) and 「項 10.4」.

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

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** [-x4] If this exists, it is installed into **etc/init/package.conf** in the package build directory. (deprecated)

See **dh_installinit**(1).

upstream/metadata [-x1] Per-package machine-readable metadata about upstream (DEP-12). See 「Upstream MEtadata GAthered with YAml (UMEGAYA)」.

Chapter 7

Quality of packaging

The quality of Debian packaging can be improved by using testing tools.

- **lintian**(1)
- **piuparts**(1)
- **autopkgtest**(1)

If you follow 「第 4 章」, these are automatically executed. You are expected to fix all warnings.

7.1 Reformat debian/* files with wrap-and-sort

It is a good idea to reformat **debian/*** files consistently using the **wrap-and-sort**(1) command in **devscripts** package.

Reformat debian/* files

```
[debhello-0.0] $ wrap-and-sort -vast
```

7.2 Validate debian/* files with debputy

The new debputy tool [1] includes subcommands to validate (and fix) most files in **debian/***.

Check correctness of files in debian/*

```
[debhello-0.0] $ debputy lint --spellcheck
```

Format debian/control and debian/tests/control files

```
[debhello-0.0] $ debputy reformat --style black
```

Using the 「**debputy reformat**」command obsoletes using 「**wrap-and-sort -vast**」.

The debputy tool also includes a language server. You can set up to get real-time feedback while editing **debian/*** files with any modern editor supporting the Language Server Protocol.

7.3 Check packaging with cme

It is a good idea to check **dpkg** configuration files using the **cme**(1) command in **cme** package. This is used by the DFSG, Licensing & New Packages Team.

Check correctness using in cme

```
[debhello-0.0] $ cme fix --verbose dpkg
```

1The main purpose of the debputy tool is to offer a new Debian package build paradigm. This new paradigm is beyond the scope of this tutorial.

Chapter 8

Sanitization of the source

There are a few cases that require sanitizing the source to prevent contamination of the generated Debian source package.

- Non- DFSG compliant content in the upstream source.
  - Debian takes software freedom seriously and adheres to the DFSG.
- Extraneous auto-generated content in the upstream source.
  - Debian packages should rebuild these under the latest system.
- Extraneous VCS content in the upstream source.
  - The **-i** and **-I** options set in 「項 4.6」for the **dpkg-source**(1) command should avoid these.
    - The **-i** option is intended for non-native Debian packages.
    - The **-I** option is intended for native Debian packages.

There are several methods to avoid including undesirable content.

8.1 Fix with Files-Excluded

This method is suitable for avoiding non-https://www.debian.org/social_contract.html#guidelines[DFSG] compliant content in the upstream source tarball.

- 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.
- **mk-origtargz** invoked from **uscan** removes excluded files from the upstream tarball and repack it as a clean tarball.
- The resulting tarball has the version number with an additional suffix **+dfsg**.

See 「**COPYRIGHT FILE EXAMPLES**」in **mk-origtargz**(1).

8.2 Fix with 「debian/rules clean」

This method is suitable for avoiding auto-generated files by removing them in the ”**debian/rules clean**” target.

注意



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

8.3 Fix with extend-diff-ignore

This is for the non-native Debian package.

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

debian/source/options to exclude the config.sub, config.guess and Makefile files:

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

注意



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

ティップ



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

8.4 Fix with tar-ignore

This is for the native Debian package.

You can exclude certain files in the source tree from the generated tarball by adjusting the file glob. Add the "**tar-ignore=**⋯" lines in the **debian/source/options** or **debian/source/local-options** files.

注意



For example, if the source package of a native package needs files with the **.o** extension as part of the test data, the setting in 「項 4.6」may be too aggressive. You can work around this by dropping the **-I** option for **DEBUILD_DPKG_BUILDPACKAGE_OPTS** in 「項 4.6」and adding the "**tar-ignore=**⋯" lines in the **debian/source/local-options** file for each package.

8.5 Fix with「git clean -dfx」

The problem of extraneous content in the second build can be avoided by restoring the source tree. This is done by committing the source tree to the Git repository before the first build.

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

```
[debhello] $ git reset --hard
[debhello] $ git clean -dfx
```

This works because the **dpkg-source** command ignores the contents of typical VCS files in the source tree, as specified by the **DEBUILD_DPKG_BUILDPACKAGE_OPTS** setting in 「項 4.6」.

ティップ



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

Chapter 9

More on packaging

Let's explore more fundamentals of Debian packaging.

9.1 Package customization

All customization data for the Debian source package resides in the **debian/** directory as presented in 「項 5.7」:

- The Debian package build system can be customized through the **debian/rules** file (see 「項 9.2」).
- 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 「項 6.14」).

When these are not sufficient to make a good Debian package, **-p1** patches of **debian/patches/*** files are deployed to modify the upstream source. These are applied in the sequence defined in the **debian/patches/series** file before building the package as presented in 「項 5.9」.

You should address the root cause of the Debian packaging problem in the least invasive way possible. This approach will make the generated package more robust for future upgrades.

注意



If the patch addressing the root cause is useful to the upstream project, send it to the upstream maintainer.

9.2 Customized debian/rules

Flexible customization of the 項 6.5 is achieved by adding appropriate **override_dh**_* targets and their rules.

When a special operation is required for a certain **dh**_*foo* command invoked by the **dh** command, its automatic execution 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**,
- 「**python -m build**」, or
- **Build.PL**.

In this case, you should add the **override_dh_auto_build** target with 「**dh_auto_build --** *arguments*」. This ensures that *arguments* are passed to the build system after the default parameters that **dh_auto_build** usually passes.

ティップ



Avoid executing bare build system commands directly if they are supported by the **dh_auto_build** command.

See:

- 「項 5.7」for the basic example.
- 「項 10.3」to be reminded of the challenge involving the underlying build system.
- 「項 10.10」for multiarch customization.
- 「項 10.6」for hardening customization.

9.3 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 Set **DEB_SOURCE**, **DEB_VERSION**, **DEB_VERSION_EPOCH_UPSTREAM**, **DEB_VERSION_UPST DEB_VERSION_UPSTREAM**, and **DEB_DISTRIBUTION** variables obtained from **dpkg-parsechangelog**(1). (useful for backport support etc..)

vendor.mk Set **DEB_VENDOR** and **DEB_PARENT_VENDOR** variables; and **dpkg_vendor_derives_from** macro obtained from **dpkg-vendor**(1). (useful for vendor support (Debian, Ubuntu, ⋯).)

architecture.mk Set **DEB_HOST**_* and **DEB_BUILD**_* variables obtained from **dpkg-architecture**(1).

buildflags.mk Set **CFLAGS**, **CPPFLAGS**, **CXXFLAGS**, **OBJCFLAGS**, **OBJCXXFLAGS**, **GCJFLAGS**, **FFLAGS**, **FCFLAGS**, and **LDFLAGS** build flags obtained from **dpkg-buildflags**(1).

For example, you can add an extra option to **CONFIGURE_FLAGS** for **linux-any** target architectures by adding the following 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
```

See 「項 10.10」, **dpkg-architecture**(1) and **dpkg-buildflags**(1).

9.4 アップストリームソフトウェアの新版更新

When a new upstream release tarball **debhello-***newvwesion***.tar.xz** is released, the Debian source package can be updated by invoking commands in the old source tree as:

```
[debhello-0.0] $ uscan
 ... debhello-newversion.tar.xz downloaded
[debhello-0.0] $ uupdate -v newversion ../debhello-newversion.tar.xz
```

- The **debian/watch** file in the old source tree must be a valid one.
- This make symlink **../debhello**_*newvwesion***.orig.tar.xz** pointing to **../debhello-***newvwesion***.tar.xz**.

- Files are extracted from **../debhello-***newvwesion***.tar.xz** to **../debhello-***newversion***/**
- Files are copied from **../debhello-***oldversion***/debian/** to **../debhello-***newvesion***/debian/** .

After the above, you should refresh **debian/patches/*** files (see「項 9.5」) and update **debian/changelog** with the **dch**(1) command.

When「**debian uupdate**」is specified at the end of line in the **debian/watch** file, **uscan** automatically executes **uupdate**(1) after downloading the tarball.

9.5 Manage patch queue with dquilt

You can add, drop, and refresh **debian/patches/*** files with **dquilt** to manage patch queue.

- **Add** a new patch **debian/patches/***bugname.patch* recording the upstream source modification on the file *buggy_file* as:

```
[debhello-0.0] $ dquilt push -a
[debhello-0.0] $ dquilt new bugname.patch
[debhello-0.0] $ dquilt add buggy_file
[debhello-0.0] $ vim buggy_file
  ...
[debhello-0.0] $ dquilt refresh
[debhello-0.0] $ dquilt header -e
[debhello-0.0] $ dquilt pop -a
```

- **Drop** (== disable) an existing patch
  - Comment out pertinent line in **debian/patches/series**
  - Erase the patch itself (optional)
- **Refresh debian/patches/*** files to make 「**dpkg-source -b**」work as expected after updating a Debian package to the new upstream release.

```
[debhello-0.0] $ uscan; uupdate  # updating to the new upstream release
[debhello-0.0] $ while dquilt push; do dquilt refresh ; done
[debhello-0.0] $ dquilt pop -a
```

  - If conflicts are encountered with 「**dquilt push**」in the above, resolve them and run 「**dquilt refresh**」manually for each of them.

9.6 Build commands

Here is a recap of popular low level 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)
- **schroot** = core of the Debian chroot environment tool
- **sbuild** = **dpkg-buildpackage** on custom **schroot** (build in the chroot)

9.7 Note on sbuild

The **sbuild**(1) command is a wrapper script of **dpkg-buildpackage** which builds Debian binary packages in a chroot environment managed by the **schroot**(1) command. For example, building for Debian **unstable** suite can be done as:

```
[debhello-0.0] $ sudo sbuild -d unstable
```

In **schroot**(1) terminology, this builds a Debian package in a clean ephemeral chroot「**chroot:unstable-amd64-sbuild**」started as a copy of the clean minimal persistent chroot 「**source:unstable-amd64-sbuild**」.

This build environment was set up as described in「項 4.7」with「**sbuild-debian-developer-setup** -s unstable」which essentially did the following:

```
[~] $ sudo mkdir -p /srv/chroot/dist-amd64-sbuild
[~] $ sudo sbuild-createchroot unstable /srv/chroot/unstable-amd64-sbuild http:// ↩
    deb.debian.org/debian
[~] $ sudo usermod -a -G sbuild <your_user_name>
[~] $ sudo shutdown -r now
 ... reboot
```

The **schroot**(1) configuration for **unstable-amd64-sbuild** was generated at **/etc/schroot/chroot.d/unstable-amd64-sbuild**.*$suffix* :

```
[unstable-amd64-sbuild]
description=Debian sid/amd64 autobuilder
groups=root,sbuild
root-groups=root,sbuild
profile=sbuild
type=directory
directory=/srv/chroot/unstable-amd64-sbuild
union-type=overlay
```

Here:

- The profile defined in the **/etc/schroot/sbuild/** directory is used to setup the chroot environment.
- **/srv/chroot/unstable-amd64-sbuild** directory holds the chroot filesystem.
- **/etc/sbuild/unstable-amd64-sbuild** is symlinked to **/srv/chroot/unstable-amd64-sbuild** .

You can update this source chroot 「**source:unstable-amd64-sbuild**」by:

```
[~] $ sudo sbuild-update -udcar unstable
```

You can log into this source chroot 「**source:unstable-amd64-sbuild**」by:

```
[~] $ sudo sbuild-shell unstable
```

ティップ



If your source chroot filesystem is missing packages such as **libeatmydata1**, **ccache**, and **lintian** for your needs, you may want to install these by logging into it.

9.8 Special build cases

The **orig.tar.xz** file may need to be uploaded for a Debian revision other than **0** or **1** under some exceptional cases (e.g., for a security upload).

When an essential package becomes a non-essential one (e.g., **adduser**), you need to remove it manually from the existing chroot environment for its use by **piuparts**.

9.9 Upload orig.tar.xz

When you first upload the package to the archive, you need to include the original **orig.tar.xz** 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**
- **sbuild --debbuildopts=-sa**
- **gbp buildpackage -sa**

ティップ



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

ティップ



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

9.10 スキップされたアップロード

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*
- **sbuild --debbuildopts=-v***1.2*
- **gbp buildpackage -v***1.2*

9.11 Bug reports

The **reportbug**(1) 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 (**README.developers**)」

ティップ



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

Chapter 10

上級パッケージング

Let's describe advanced topics on Debian packaging.

10.1 Historical perspective

Let me oversimplify historical perspective of Debian packaging practices focused on the non-native packaging.

Debian was started in 1990s when upstream packages were available from public FTP sites such as Sunsite. In those early days, Debian packaging used Debian source format currently known as the Debian source format 「**1.0**」:

- The Debian source package ships a set of files for the Debian source package.
  - *package_version*.**orig.tar.xz** : symlink to or copy of the upstream released file.
  - *package_version-revision*.**diff.gz** : 「**One big patch**」for Debian modifications.
  - *package_version-revision*.**dsc** : package description.
- Several workaround approaches such as **dpatch**, **dbs**, or **cdbs** were deployed to manage multiple topic patches.

The modern Debian source format「**3.0 (quilt)**」was invented around 2008 (see「ProjectsDebSrc3.0」):

- The Debian source package ships a set of files for the Debian source package.
  - *package_version*.**orig.tar.?z** : symlink to or copy of the upstream released file.
  - *package_version-revision*.**debian.tar.?z** : tarball of **debian/** for Debian modifications.
    - The **debian/source/format** file contains 「**3.0 (quilt)**」.
    - Optional multiple topic patches are stored in the **debian/patches/** directory.
  - *package_version-revision*.**dsc** : package description.
- The standardized approach to manage multiple topic patches using **quilt**(1) is deployed for the Debian source format 「**3.0 (quilt)**」.

Most Debian packages adopted the Debian source formats 「**3.0 (quilt)**」and 「**3.0 (native)**」.

Now, the **git**(1) is popular with upstream and Debian developers. The **git** and its associated tools are important part of the modern Debian packaging workflow. This modern workflow involving **git** will be mentioned later in 「第 11 章」.

10.2 Current trends

Current Debian packaging practices and their trends are moving target. See:

- 「Debian Trends」 — Hints for 「De facto standard」of Debian practices

  - Build systems: **dh**
  - Debian source format: 「**3.0 (quilt)**」
  - VCS: **git**
  - VCS Hosting: salsa
  - Rules-Requires-Root: adopted, fakeroot
  - Copyright format: DEP-5

- 「**debhelper-compat-upgrade-checklist**(7) manpage」 — Upgrade checklist for **debhelper**
- 「DEP - Debian Enhancement Proposals」 — Formal proposals to enhance Debian

You can also search entire Debian source code data by yourself, too.

- 「Debian Sources」 — code search tool
  - 「Debian Code Search」 — wiki page describing its usage
- 「Debian Code Search」 — another code search tool

10.3 Note on build system

Auto-generated files of the build system may be found in the released upstream tarball. These should be regenerated when Debian package is build. E.g.:

- 「**dh $@ --with autoreconf**」 should be used in the **debian/rules** if Autotools (**autoconf** + **automake**) are used.

Some modern build system may be able to download required source codes and binary files from arbitrary remote hosts to satisfy build requirements. Don't use this download feature. The official Debian package is required to be build only with packages listed in **Build-Depends:** of the **debian/control** file.

10.4 Continuous integration

The **dh_auto_test**(1) command is a **debhelper** command that tries to automatically run the test suite provided by the upstream developer during the Debian package building process.

The **autopkgtest**(1) command can be used after the Debian package building process. It tests generated Debian binary packages in the virtual environment using the **debian/tests/control** RFC822-style metadata file as continuous integration (CI). See:

- Documents in the **/usr/share/doc/autopkgtest/** directory
- 「4. autopkgtest: Automatic testing for packages」 of the 「Ubuntu Packaging Guide」

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

- The Salsa offers 「項 11.3」.
- The **debci** package: CI platform on top of the **autopkgtest** package
- The **jenkins** package: generic CI platform

10.5 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 **DEB_BUILD_PROFILES** environment variable.

See Debian wiki: 「BuildProfileSpec」.

ティップ



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.

10.6 Compiler hardening

The compiler hardening support spreading for Debian **jessie** (8.0) 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_OP DEB_CFLAGS_MAINT_APPEND**, and **DEB_LDFLAGS_MAINT_APPEND** (see 「第 5 章」and **dpkg-buildflags**(1)).

10.7 Reproducible build

Here are some recommendations to attain a reproducible build result.

- Don't embed the timestamp based on the system time.
- Don't embed the file path of the build environment.
- Use 「**dh $@**」in the **debian/rules** to access the latest **debhelper** features.
- Export the build environment as 「**LC_ALL=C.UTF-8**」(see 「項 12.1」).
- Set the timestamp used in the upstream source from the value of the debhelper-provided environment variable **$SOURCE_DATE_EPOCH**.
- Read more at 「ReproducibleBuilds」.
  - 「ReproducibleBuilds Howto」.
  - 「ReproducibleBuilds TimestampsProposal」.

Reproducible builds are important for security and quality assurance. They allow independent verification that no vulnerabilities or backdoors have been introduced during the build process.

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

10.8 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.

10.9 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 「項 10.17」.
- Debian ships shared libraries such as **/usr/lib/<triplet>/lib***foo-0.1***.so.***1.0.0* (see 「項 10.10」).
- Debian encourages using versioned symbols in the shared library (see 「項 10.16」).
- 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」 1
  - This encourages having versioned symbols in the shared library.
- 「Debian Library Packaging guide」 2
  - Please read the discussion thread following its announcement, too.

1This document was written before the introduction of the **symbols** file.

2The 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.

10.10 マルチアーチ

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:

- 「項 9.3」
- 「項 16.2」
- 「項 10.12」
- 「**dpkg-architecture**(1) manpage」

10.11 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:

- 「項 14.11」(Autotools based)
- 「項 14.12」(CMake based)

An intuitive and flexible method to create the initial template **debian/control** file defining the split of the Debian binary packages is accommodated with the **-b** option. See 「項 16.2」.

10.12 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.xz**
- a tool source *bar-1.0***.tar.xz** written in a compiled language
- a tool source *baz-1.0***.tar.xz** written in an interpreted language

binarypackage	***type***	**Architecture:**	**Multi-Arch:**	パッケージ内容
lib*foo1*	**lib***	any	same	共有ライブラリー、同時インストール可能
lib*foo***-dev**	**dev***	any	same	共有ライブラリーヘッダーファイル他、同時インストール可能
lib*foo***-tools**	**bin***	any	foreign	実行時サポートプログラム、同時インストール不可
lib*foo***-doc**	**doc***	all	foreign	共有ライブラリーのドキュメンテーションファイル
bar	**bin***	any	foreign	コンパイルされ実行されるプログラムファイル、同時インストール不可
*bar***-doc**	**doc***	all	foreign	プログラムのドキュメンテーションファイル
baz	**script**	all	foreign	インタープリターで実行されるプログラムファイル

10.13 Multiarch library path

Debian policy requires to comply with the 「Filesystem Hierarchy Standard (FHS), version 3.0」, with the exceptions noted in 「File System Structure」.

The most notable exception is the use of **/usr/lib/<triplet>/** instead of **/usr/lib<qual>/** (e.g., **/lib32/** and **/lib64/**) to support a multiarch library.

Table 10.2 The multiarch library path options

Classic path	**i386** マルチアーチパス	**amd64** マルチアーチパス
/lib/	**/lib/i386-linux-gnu/**	**/lib/x86_64-linux-gnu/**
/usr/lib/	**/usr/lib/i386-linux-gnu/**	**/usr/lib/x86_64-linux-gnu/**

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.

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.

10.14 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 10.3 The multiarch header file path options

Classic path	**i386** マルチアーチパス	**amd64** マルチアーチパス
/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>/**. 3

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

10.15 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 10.4 The ***.pc** file path options

Classic path	**i386** マルチアーチパス	**amd64** マルチアーチパス
/usr/lib/pkgconfig/	**/usr/lib/i386-linux-gnu/pkgconfig/**	**/usr/lib/x86_64-linux-gnu/pkgconfig/**

10.16 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.

3This 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.」

        * 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.

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

You should also check:

- Debian wiki: 「UsingSymbolsFiles」
- Debian wiki: 「Projects/ImprovedDpkgShlibdeps」
- Debian kde team: 「Working with symbols files」
- 「項 14.11」
- 「項 14.12」

ティップ



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.

10.17 Library package name

Let's consider that the upstream source tarball of the **lib***foo* library is updated from **lib***foo-7.0***.tar.xz** to **lib***foo-8.0***.tar.xz** 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.xz**.

警告



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** suite 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.xz** and **lib***foo8-8.0***.tar.xz** respectively from **lib***foo-?.0***.tar.xz**.
    * 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 (Fails To Build From Source).

ティップ



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

See 「項 10.9」.

10.18 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」.

注意



Please make sure to rename binary packages as in 「項 10.17」.

10.19 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:Version}.0~)**

10.20 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) manapage」
- 「**strip**(1) manapage」
- 「**readelf**(1) manapage」
- 「**objcopy**(1) manapage」
- Debian wiki: 「DebugPackage」
- Debian wiki: 「AutomaticDebugPackages」
- Debian debian-devel post: 「Status on automatic debug packages」(2015-08-15)

10.21 -dbgsym package

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.

- The **debian/rules** file shouldn't explicitly contain **dh_strip**.
- Set the **Build-Depends** to **debhelper-compat (>=14)** while removing **Build-Depends** to **debhelper** in **debian/control**.

10.22 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.

These **debconf** files are called by package configuration scripts in the binary Debian package

- **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」.

Chapter 11

Packaging with git

Up to 「第 10 章」, we focused on packaging operations without using Git or any other VCS. These traditional packaging operations were based on the tarball released by the upstream as mentioned in 「項 10.1」.

Currently, the **git**(1) command is the de-facto platform for the VCS tool and is the essential part of both upstream development and Debian packaging activities. (See Debian wiki 「Debian git packaging maintainer branch formats and workflows」 for existing VCS workflows.)

注意



Since the non-native Debian source package uses「**diff -u**」as its backend technology for the maintainer modification, it can't represent modification involving symlink, file permissions, nor binary data (March 2022 discussion on debian-devel@l.d.o). Please avoid making such maintainer modifications even though these can be recorded in the Git repository.

Since VCS workflows are complicated topic and there are many practice styles, I only touch on some key points with minimal information, here.

Salsa is the remote Git repository service with associated tools. It offers the collaboration platform for Debian packaging activities using a custom GitLab application instance. See:

- 「項 11.1」
- 「項 11.2」
- 「項 11.3」

There are 2 styles of branch names for the Git repository used for the packaging. See 「項 11.4」.
There are 2 main usage styles for the Git repository for the packaging. See:

- 「項 11.5」
- 「項 11.13」

There are 2 notable Debian packaging tools for the Git repository for the packaging.

- **gbp**(1) and its subcommands:
  - This is a tool designed to work mainly with 「項 11.5」.
  - See 「項 11.9」.
- **dgit**(1) and its subcommands:
  - This is a tool designed to work mainly with 「項 11.13」.
  - This contains a tool to upload Debian packages using the **dgit** server.
  - See 「項 11.14」.

11.1 Salsa repository

It is highly desirable to host Debian source code package on Salsa. Over 90% of all Debian source code packages are hosted on Salsa. 1

The exact VCS repository hosting an existing Debian source code package can be identified by a metadata field `Vcs-*` which can be viewed with the `apt-cache showsrc <package-name>` command.

11.2 Salsa account setup

After signing up for an account on Salsa, make sure that the following pages have the same e-mail address and openPGP keys you have configured to be used with Debian, as well as your SSH key:

- https://salsa.debian.org/-/profile/emails
- https://salsa.debian.org/-/user_settings/gpg_keys
- https://salsa.debian.org/-/user_settings/ssh_keys

11.3 Salsa CI service

Salsa runs Salsa CI service as an instance of GitLab CI for 「項 10.4」.

For every 「**git push**」instances, tests which mimic tests run on the official Debian package service can be run by setting Salsa CI configuration file 「項 6.13」as:

```
---
include:
  - https://salsa.debian.org/salsa-ci-team/pipeline/raw/master/recipes/debian.yml

# Customizations here
```

11.4 Branch names

The Git repository for the Debian packaging should have at least 2 branches:

- **debian-branch** to hold the current Debian packaging head.
  - old style: **master** (or **debian**, **main**, ⋯)
  - DEP-14 style: **debian/latest**
- **upstream-branch** to hold the upstream releases in the imported form.
  - old style: **upstream**
  - DEP-14 style: **upstream/latest**

In this tutorial, old style branch names are used in examples for simplicity.

注意



This **upstream-branch** may need to be created using the tarball released by the upstream independent of the upstream Git repository since it tends to contain automatically generated files.

The upstream Git repository content can co-exit in the local Git repository used for the Debian packaging by adding its copy. E.g.:

1Use of git.debian.org or alioth.debian.org are deprecated now.

```
[debhello] $ git remote add upstreamvcs <url-upstream-git-repo>
[debhello] $ git fetch upstreamvcs master:upstream-master
```

This allows easy cherry-picking from the upstream Git repository for bug fixes.

11.5 Patch unapplied Git repository

The patch unapplied Git repository can be summarized as:

- This seems to be the traditional practice as of 2024.
- The source tree matches extracted contents by 「**dpkg-source -x --skip-patches**」of the Debian source package.
  - The upstream source is recorded in the Git repository without changes.
  - The maintainer modified contents are confined within the **debian/*** directory.
  - Maintainer changes to the upstream source are recorded in **debian/patches/*** files for the Debian source format 「**3.0 (quilt)**」.
- This repository style is useful for all variants of traditional workflows and **gbp** based workflow:
  - 「項 5.7」(no patch)
  - 「項 5.10」
    - **debian/patches/*** files can also be generated using 「**git format-patch**」, 「**git diff**」, or 「**gitk**」from **git** commits in the through-away maintainer modification branch or from the upstream unreleased commits.
  - 「項 5.11」including the last 「**dquilt pop -a**」step
  - 「項 11.6」
- Use helper scripts such as **dquilt**(1) and **gbp-pq**(1) to manage data in **debian/patches/*** files.
  - Add **.pc** line to the **~/.gitignore** file if **dquilt** is used.
- Use 「**dpkg-source -b**」to build the Debian source package.
- Use **dput**(1) to upload the Debian source package.

11.6 Patch by 「gbp-pq」approach

For「項 11.5」, you can generate **debian/patches/*** files using the **gbp-pq**(1) command from **git** commits in the through-away **patch-queue** branch.

Unlike **dquilt** which offers similar functionality as seen 「項 5.11」and 「項 9.5」, **gbp-pq** doesn't use **.pc/*** files to track patch state, but instead **gbp-pq** utilizes temporary branches in git.

11.7 Manage patch queue with gbp-pq

You can add, drop, and refresh **debian/patches/*** files with **gbp-pq** to manage patch queue.

If the package is managed in 「項 11.5」using the **git-buildpackage** package, you can revise the upstream source to fix bug as the maintainer and release a new Debian revision using **gbp pq**.

- **Add** a new patch recording the upstream source modification on the file *buggy_file* as:

```
[debhello] $ git checkout master
[debhello] $ gbp pq import
gbp:info: ... imported on 'patch-queue/master
[debhello] $ vim buggy_file
  ...
[debhello] $ git add buggy_file
```

```
[debhello] $ git commit
[debhello] $ gbp pq export
gbp:info: On 'patch-queue/master', switching to 'master'
gbp:info: Generating patches from git (master..patch-queue/master)
[debhello] $ git add debian/patches/*
[debhello] $ dch -i
[debhello] $ git commit -a -m "Closes: #<bug_number>"
[debhello] $ git tag debian/<version>-<rev>
```

- **Drop** (== disable) an existing patch
  - Comment out pertinent line in **debian/patches/series**
  - Erase the patch itself (optional)
- **Refresh debian/patches/*** files to make 「**dpkg-source -b**」work as expected after updating a Debian package to the new upstream release.

```
[debhello] $ git checkout master
[debhello] $ gbp pq --force import # ensure patch-queue/master branch
gbp:info: ... imported on 'patch-queue/master
[debhello] $ git checkout master
[debhello] $ gbp import-orig --pristine-tar --uscan
  ...
gbp:info: Successfully imported version ?.?.? of ../packagename_?.?.?.orig. ↩
    tar.xz
[debhello] $ gbp pq rebase
 ... resolve conflicts and commit to patch-queue/master branch
[debhello] $ gbp pq export
gbp:info: On 'patch-queue/master', switching to 'master'
gbp:info: Generating patches from git (master..patch-queue/master)
[debhello] $ git add debian/patches
[debhello] $ git commit -m "Update patches"
[debhello] $ dch -v <newversion>-1
[debhello] $ git commit -a -m "release <newversion>-1"
[debhello] $ git tag debian/<newversion>-1
```

11.8 gbp import-dscs --debsnap

For Debian source packages named「*<source-package>*」recorded in the snapshot.debian.org archive, an initial git repository managed in「項 11.5」with all of the Debian version history can be generated as follows.

```
[debhello] $ gbp import-dscs --debsnap --pristine-tar <source-package>
```

11.9 Note on gbp

The **gbp** command is provided by the **git-buildpackage** package.

- This command is designed to manage contents of 「項 11.5」efficiently.
- Use 「**gbp import-orig**」to 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.
- Use 「**gbp import-dsc**」to import the previous Debian source package to the git repository.
- Use 「**gbp dch**」to generate the Debian changelog from the git commit messages.

- Use 「**gbp buildpackage**」 to build the Debian binary package from the git repository.
  - The **sbuild** package can be used as its clean chroot build backend either by configuration or adding 「**--git-builder='sbuild -A -s --source-only-changes -v -d unstable'**」
- Use 「**gbp pull**」 to update the **debian**, **upstream** and **pristine-tar** branches safely from the remote repository.
- Use 「**gbp pq**」 to manage quilt patches without using **dquilt** command.
- Use 「**gbp clone** *REPOSITORY_URL*」 to clone and set up tracking branches for **debian**, **upstream** and **pristine-tar**.

Package history management with the **git-buildpackage** package is becoming the standard practice for many Debian maintainers. See more at:

- 「Building Debian Packages with git-buildpackage」
- 「4 tips to maintain a "3.0 (quilt)"Debian source package in a VCS」
- The **systemd** packaging practice documentation on 「Building from source」
- The workflow mentioned in **dgit-maint-gbp**(7) which enables to use this **gbp** with **dgit**

11.10 The Git repository browser

The **gitk** command in the **gitk** package displays changes in a repository or a selected set of commits. This includes visualizing the commit graph, showing information related to each commit, and the files in the trees of each revision.

This **gitk** command also provides very intuitive UI to many cumbersome operations of the 「**git**」 command such as 「**git checkout** ⋯」, 「**git reset*** ⋯」, 「**git diff** ⋯」, etc..

11.11 Git commit history organization

When your local Git commit history becomes intertwined, you need to organize it before pushing it out to the public.

The most simple organization process is to squash all changes to a single commit using 「**git rebase -i**」 interactively.

But this may create a huge commit with files such as auto-generated files not intended to be committed. You can **drop** such files in the commit using 「**git rm** *some_file*」 and 「**git commit --amend**」. This may be quite cumbersome.

This cumbersome **drop** process can be eased by using the 「**git-ime**」 command in the **imediff** package. It automatically splits a single commit with many files into multiple commits involving only a single file changes. Now you can drop such files using 「**git rebase -i**」 interactively.

ティップ



The 「**git-ime**」 operating on a single file change commit splits it into multiple commits of line changes using **imediff** interactively. Invoking this with the **--auto** option will automate this split commit operation. See **git-ime**(1) and **imediff**(1).

11.12 Quasi-native Debian packaging

The **quasi-native** packaging scheme packages a source without the real upstream tarball using the **non-native** package format.

ティップ



Some people promote this **quasi-native** packaging scheme even for programs written only for Debian since it helps to ease communication with the downstream distros such as Ubuntu for bug fixes etc.

This **quasi-native** packaging scheme involves 2 preparation steps:

- Organize its source tree almost like **native** Debian package (see「項 6.4」) with **debian/*** files with a few exceptions:
  - Make **debian/source/format** to contain 「**3.0 (quilt)**」instead of 「**3.0 (native)**」.
  - Make **debian/changelog** to contain *version*-*revision* instead of *version* .
- Generate missing upstream tarball preferably without **debian/*** files.
  - For Debian source format「**3.0 (quilt)**」, removal of files under **debian/** directory is an optional action.

The rest is the same as the **non-native** packaging workflow as written in 「項 6.1」.
Although this can be done in many ways, you can use the Git repository and 「**git deborig**」as:

```
[~] $ cd /path/to/debhello
[debhello] $ dch -r
  ... set its <version>-<revision>, e.g., 1.0-1
[debhello] $ git tag -s debian/1.0-1
[debhello] $ git rm -rf debian
[debhello] $ git tag -s upstream/1.0
[debhello] $ git commit -m upstream/1.0
[debhello] $ git reset --hard HEAD^
[debhello] $ git deborig
[debhello] $ sbuild
```

11.13 Patch applied Git repository

注意



The focus of this introductory tutorial 「Guide for Debian Maintainers」isn't the patch applied Git repository which is rather a new trend initiated by the proponent of the **dgit** command. So minimal explanation is given here.

The patch applied Git repository can be summarized as:

- The source tree matches extracted contents by「**dpkg-source -x**」of the Debian source package.
  - The source tree is buildable and the same as what NMU maintainers see.
  - The source is recorded in the Git repository with maintainer changes including the **debian/** directory.
  - Maintainer changes to the upstream source are also recorded in **debian/patches/*** files for the Debian source format 「**3.0 (quilt)**」.

11.14 Note on dgit

The **dgit** command is provided by the **dgit** package.

- This command enables to access the Debian package repository as if it were a **git** remote repository.
- This command offers tools to manage Debian packaging activities mainly using 「項 11.13」.
  - No more convoluted operations to manage patch files in the **debian/patches** directory.
- Use 「**dgit build-source**」or 「**dgit sbuild**」to build the Debian source-only or source+binary package.
- Use「**dgit push-source**」or「**dgit push-build**」for uploading the Debian source-only or source+binary package via the **dgit** server.
- Use **git-deborig**(1) to produce Debian *package***.orig.tar.xz** from the upstream version in **debian/changelog**.

ティップ



The **dgit** server is browsable at https://browse.dgit.debian.org/ site.

注意



In order to keep the working tree **dgit**-compatible, delete **debian/source/local-options** and **debian/source/local-patch-header** if they exist.

Hints for workflow styles:

- **dgit-maint-merge**(7) workflow.
  - Use this for the Debian non-native package without granular topic patches recorded in the Debian source package.
    - Good enough for packages only with trivial modifications to the upstream.
    - Only choice for packages with intertwined modification histories to the upstream.
  - Add **auto-commit** and **single-debian-patch** lines in the **debian/source/options** file
    - No granular topic patches recorded inside of the Debian source package.
  - Use 「**git checkout upstream; git pull**」to pull the new upstream commit and use 「**git checkout master ; git merge <new-version-tag>**」to merge it to the **master** branch.
  - See 「項 5.12」for example.
- **dgit-maint-debrebase**(7) workflow.
  - Use this for the Debian non-native package with granular topic patches recorded in the Debian source package.
  - Use the **git-debrebase**(1) command to maintain series of Debian changes to upstream source.
- **dgit-maint-native**(7) workflow,
  - Use this for the Debian native package in the Debian Git repository. (No maintainer changes)
- **dgit-maint-gbp**(7) workflow

  - Use this for the Debian non-native package using source format「**3.0 (quilt)**」with its Debian Git repository which had been using **gbp-buildpackage**(1) with「項 11.5」.

This author likes this new **dgit** command and just started to use it with **dgit-maint-merge**(7) and **dgit-maint-native**(7) workflows. Thus, topics around **dgit** are beyond this tutorial document to cover in depth. Please start reading the latest relevant manpages and upstream resources:

- 「dgit: use the Debian archive as a git remote (2015)」
- 「tag2upload (2023)」

Chapter 12

Tips

Please also read insightful pages linked from 「Notes on Debian」 by Russ Allbery (long time Debian developer) which have best practices for advanced packaging topics.

12.1 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
```

12.2 UTF-8 変換

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 the encoding of plain text files.

```
[debhello] $ 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.

```
[debhello] $ 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.

12.3 Hints for Debugging

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.

- Wikipedia: 「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
- 「**python3 -m py_compile** *script.py*」- Syntax check of a Python script
- 「**python3 -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**」

ティップ



The **script** command records console outputs.

ティップ



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

ティップ



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.

ティップ



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

Chapter 13

Tool usages

Here are some notable tools around Debian packaging.

注意



The descriptions in this section are intentionally brief. Prospective maintainers are strongly encouraged to search for and read all relevant documentation associated with these commands.

注意



Examples here use the **gz**-compression. The **xz**-compression may be used instead.

13.1 debdiff

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

```
[base_dir] $ debdiff old-package.dsc new-package.dsc
```

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

```
[base_dir] $ debdiff old-package.changes new-package.changes
```

これらのコマンドは、ソースパッケージ中でどんな変更をしたのか、バイナリーパッケージの中で意図せず削除したり配置を間違えたりしていないか、そしてバイナリーパッケージの更新時に不用意な変更をしていないかどうか、といった事柄を確認するのに便利です。

Debian now enforces the source-only upload when developing packages. So there may be 2 different ***.changes** files:

- *package_version-revision*_**source.changes** for the normal source-only upload
- *package_version-revision_arch*.**changes** for the source+binary upload

13.2 dget

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

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

13.3 mk-origtargz

You can make the upstream tarball **../foo-***newversion***.tar.[xg]z** accessible from the Debian source tree as **../foo**_*newversion*.**orig.tar.[xg]z**. This command is useful for renaming and symlinking the upstream tarball to the expected Debian naming convention.

13.4 origtargz

You can fetch the pre-existing orig tarball of a Debian package from various sources, and unpack it with **origtargz** command.

This is basically for **-2**, **-3**, ⋯revisions.

注意



When the upstream tarball is missing, **debmake** automatically produces a required tarball. This is a convenient feature and good enough for making a private Debian package. But when making a Debian package for the official Debian repository, you must use exactly the same upstream tarball as the **-1** revision. For such case, **origtargz** should be used.

13.5 git deborig

If the upstream project is hosted in a Git repository without an official tarball release, you can generate its orig tarball from the **git** repository for use by the Debian source package. Execute 「git deborig」 from the root of the checked-out source tree.

This is basically for **-1** revisions.

13.6 dpkg-source -b

The 「**dpkg-source -b**」 command packs the upstream source tree into the Debian source package.

It expects a series of patches in the **debian/patches/** directory and their application sequence in **debian/patches/series**.

It is compatible with **dquilt** (see 「項 4.4」) operations and understands the patch application status from the existence of **.pc/applied-patches**.

The **dpkg-buildpackage** command invokes 「**dpkg-source -b**」.

13.7 dpkg-source -x

The「**dpkg-source -x**」command extracts the source tree and applies the patches in the **debian/patches/** directory using the sequence specified in **debian/patches/series** to the upstream source tree. It also adds **.pc/applied-patches**. (See 「項 11.13」.)

The「**dpkg-source -x --skip-patches**」command extracts source tree only. It doesn't add **.pc/applied-patches**. (See 「項 11.5」.)

Both extracted source trees are ready for building Debian binary packages with **dpkg-buildpackage**, **dbuild**, **sbuild**, etc..

13.8 debc

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

```
[base_dir] $ debc package_version-rev_arch.changes
```

13.9 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
```

13.10 dpkg-depcheck

You can use **dpkg-depcheck**(1) to obtain a good first approximation to the **Build-Depends** line needed by a Debian package.

```
[foo-1.0] $ dpkg-depcheck -b debian/rules build
```

Chapter 14

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 programming topics.

- Programming 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 and C sources.
- Overview of many build systems: Makefile, Python, 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 「第 5 章」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.

ティップ



The best source of a packaging example is the current Debian archive itself. Please use the 「Debian Code Search」service to find pertinent examples.

14.1 Cherry-pick templates

Here is an example of creating a simple Debian package from a zero-content source in an empty directory.

This is a good way to obtain all the template files without cluttering the upstream source tree you are working on.

Let's assume this empty directory to be **debhello-0.1**.

```
[base_dir] $ mkdir debhello-0.1
[base_dir] $ tree
.
+-- debhello-0.1

2 directories, 0 files
```

Let's generate the maximum amount of template files.

Let's also use the 「**-p** *debhello* **-t -x3 -u** *0.1* **-r** *1*」 options to create the missing upstream tarball with optional **-x3**, and **-t** options.

```
[base_dir] $ cd debhello-0.1
[debhello-0.1] $ debmake -p debhello -x3 -t -T -u 0.1 -r 1
I: debmake (version: 5.1.5)
 ...
```

Let's inspect generated template files.

```
[debhello-0.1] $ cd ..
[base_dir] $ tree
.
+-- debhello-0.1
|   +-- debian
|       +-- README.Debian
|       +-- README.source
|       +-- bug-control.ex
|       +-- bug-presubj.ex
|       +-- bug-script.ex
|       +-- changelog
|       +-- clean
|       +-- conffiles.ex
|       +-- control
|       +-- copyright
|       +-- cron.d.ex
|       +-- cron.daily.ex
|       +-- cron.hourly.ex
|       +-- cron.monthly.ex
|       +-- cron.weekly.ex
|       +-- default.ex
|       +-- dirs
|       +-- doc-base.ex
|       +-- docs
|       +-- emacsen-install.ex
|       +-- emacsen-remove.ex
|       +-- emacsen-startup.ex
|       +-- examples
|       +-- gbp.conf
|       +-- info.ex
|       +-- install
|       +-- links
|       +-- lintian-overrides.ex
|       +-- maintscript.ex
|       +-- manpage.1.ex
|       +-- manpage.asciidoc.ex
|       +-- manpage.md.ex
|       +-- manpage.sgml.ex
|       +-- manpage.xml.ex
|       +-- manpages
|       +-- patches
|       |   +-- series
|       +-- postinst.ex
|       +-- postrm.ex
|       +-- preinst.ex
|       +-- prerm.ex
```

```
|       +-- rules
|       +-- salsa-ci.yml
|       +-- service.ex
|       +-- source
|       |   +-- format
|       |   +-- lintian-overrides.ex
|       |   +-- options.ex
|       |   +-- patch-header.ex
|       +-- tests
|       |   +-- control
|       +-- tmpfile.ex
|       +-- upstream
|       |   +-- metadata
|       +-- watch
+-- debhello-0.1.tar.xz
+-- debhello_0.1.orig.tar.xz -> debhello-0.1.tar.xz

7 directories, 53 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.

14.2 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.xz**.

This type of source has no automated means and files must be installed manually.

For example:

```
[base_dir] $ tar --xz -xmf debhello-0.2.tar.xz
[base_dir] $ cd debhello-0.2
[debhello-0.2] $ sudo cp scripts/hello /bin/hello
 ...
```

Let's get this source as tar file from a remote site and make it the Debian package.

Download debhello-0.2.tar.xz

```
[base_dir] $ wget http://www.example.org/download/debhello-0.2.tar.xz
 ...
[base_dir] $ tar --xz -xmf debhello-0.2.tar.xz
[base_dir] $ tree
.
+-- debhello-0.2
|   +-- README.md
|   +-- data
|   |   +-- hello.desktop
|   |   +-- hello.png
|   +-- man
|   |   +-- hello.1
|   +-- scripts
|       +-- hello
+-- debhello-0.2.tar.xz

5 directories, 6 files
```

Here, the POSIX shell script **hello** is a very simple one.

hello (v=0.2)

```
[base_dir] $ 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)

```
[base_dir] $ 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.

```
[base_dir] $ cd debhello-0.2
[debhello-0.2] $ debmake -b':sh' -x1
I: debmake (version: 5.1.5)
I: Copyright © 2014-2026 Osamu Aoki <osamu@debian.org>
I: [debhello-0.2] $ cd ..
I: Non-native Debian package pkg="debhello", ver="0.2", rev="1" method="dir_d...
I: already in the package-version form: "debhello-0.2"
I: [base_dir] $ ln -sf 'debhello-0.2.tar.xz' 'debhello_0.2.orig.tar.xz'
I: [base_dir] $ cd debhello-0.2
I: parsing option -b ":sh"
I: binary package=debhello Type=script / Arch=all M-A=foreign
I: build_type = Unknown
I: ext_type = 1                       1 files
I: ext_type = desktop                 1 files
I: ext_type = md                      1 files
I: creating debian/* files with "-x 1" option
I: [debhello-0.2] $ licensecheck --recursive --copyright --deb-machine  . > d...
I: creating debian/copyright by licensecheck.
I: creating debian/control from control.py
I: creating debian/control by control.py
I: creating debian/changelog from extra0_changelog
I: creating debian/rules from extra0_rules
I: creating debian/source/format from extra0source_format
 ...
```

Let's inspect notable template files generated.

The source tree after the basic debmake execution. (v=0.2)

```
[debhello-0.2] $ cd ..
[base_dir] $ tree
.
+-- debhello-0.2
|   +-- README.md
|   +-- data
|   |   +-- hello.desktop
|   |   +-- hello.png
|   +-- debian
|   |   +-- README.Debian
|   |   +-- README.source
|   |   +-- changelog
|   |   +-- clean
|   |   +-- control
```

```
|   |   +-- copyright
|   |   +-- dirs
|   |   +-- docs
|   |   +-- examples
|   |   +-- gbp.conf
|   |   +-- install
|   |   +-- links
|   |   +-- manpages
|   |   +-- patches
|   |   |   +-- series
|   |   +-- rules
|   |   +-- salsa-ci.yml
|   |   +-- source
|   |   |   +-- format
|   |   +-- tests
|   |   |   +-- control
|   |   +-- upstream
|   |   |   +-- metadata
|   |   +-- watch
|   +-- man
|   |   +-- hello.1
|   +-- scripts
|       +-- hello
+-- debhello-0.2.tar.xz
+-- debhello_0.2.orig.tar.xz -> debhello-0.2.tar.xz

10 directories, 27 files
```

debian/rules (template file, v=0.2):

```
[base_dir] $ cd debhello-0.2
[debhello-0.2] $ cat debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
# See debhelper(7) (un-comment to enable)
# This is an autogenerated template for debian/rules.
#
# Output every command that modifies files on the build system.
#export DH_VERBOSE = 1
#
# Copy some variable definitions from pkg-info.mk and vendor.mk
# under /usr/share/dpkg/ to here if they are useful.
#
# These are rarely used code. (START)
#
# The following include for *.mk magically sets miscellaneous
# variables while honoring existing values of pertinent
# environment variables:
#
# Architecture-related variables such as DEB_TARGET_MULTIARCH:
#include /usr/share/dpkg/architecture.mk
# Vendor-related variables such as DEB_VENDOR:
#include /usr/share/dpkg/vendor.mk
# Package-related variables such as DEB_DISTRIBUTION
#include /usr/share/dpkg/pkg-info.mk
#
# You may alternatively set them susing a simple script such as:
# DEB_VENDOR ?= $(shell dpkg-vendor --query  Vendor)
#
# These are rarely used code. (END)
#

### main packaging script based on post dh7 syntax
%:
        dh $@
```

```
# debmake generated override targets
```

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):

```
[debhello-0.2] $ cat debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: "Osamu Aoki" <osamu@debian.org>
Build-Depends:
 debhelper-compat (= 13),
Standards-Version: 4.7.3
Homepage: <insert the upstream URL, if relevant>
Rules-Requires-Root: no
#Vcs-Git: https://salsa.debian.org/debian/debhello.git
#Vcs-Browser: https://salsa.debian.org/debian/<project_site>

Package: debhello
Section: unknown
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.
 .
 ===== This comes from the unmodified template file =====
 .
 Please edit this template file (debian/control) and other package files
 (debian/*) to make them meet all the requirements of the Debian Policy
 before uploading this package to the Debian archive.
 .
 See
  * https://www.debian.org/doc/manuals/developers-reference/best-pkging-pract...
  * https://www.debian.org/doc/manuals/debmake-doc/ch05.en.html#control
 .
 The synopsis description at the top should be about 60 characters and
 written as a phrase.  No extra capital letters or a final period.  No
 articles b''—b'' "a", "an", or "the".
 .
 The package description for general-purpose applications should be
 written for a less technical user.  This means that we should avoid
 jargon.  GNOME or KDE is fine but GTK+ is probably not.
 .
 Use the canonical forms of words:
  * Use X Window System, X11, or X; not X Windows, X-Windows, or X Window.
  * Use GTK+, not GTK or gtk.
  * Use GNOME, not Gnome.
  * Use PostScript, not Postscript or postscript.
```

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 「第 6 章」.

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):

```
[base_dir] $ cd debhello-0.2
[debhello-0.2] $ vim debian/rules
 ... hack, hack, hack, ...
[debhello-0.2] $ cat debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1

%:
        dh $@
```

debian/control (maintainer version, v=0.2):

```
[debhello-0.2] $ vim debian/control
 ... hack, hack, hack, ...
[debhello-0.2] $ cat debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends:
 debhelper-compat (= 13),
Standards-Version: 4.7.3
Homepage: https://salsa.debian.org/debian/debmake-doc
Rules-Requires-Root: no

Package: debhello
Architecture: all
Multi-Arch: foreign
Depends:
 ${misc:Depends},
Description: Simple packaging example for debmake
 This Debian binary package is an example package.
 (This is an example only)
```

警告



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):

```
[debhello-0.2] $ vim debian/install
 ... hack, hack, hack, ...
[debhello-0.2] $ cat 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 debian/manpages
 ... hack, hack, hack, ...
[debhello-0.2] $ cat 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):

```
[debhello-0.2] $ rm -f debian/clean debian/dirs debian/links
[debhello-0.2] $ rm -f debian/README.source debian/source/*.ex
[debhello-0.2] $ rm -rf debian/patches
```

```
[debhello-0.2] $ tree -F debian
debian/
+-- README.Debian
+-- changelog
+-- control
+-- copyright
+-- docs
+-- examples
+-- gbp.conf
+-- install
+-- manpages
+-- rules*
+-- salsa-ci.yml
+-- source/
|   +-- format
+-- tests/
|   +-- control
+-- upstream/
|   +-- metadata
+-- watch

4 directories, 15 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.

```
[base_dir] $ cd debhello-0.2
[debhello-0.2] $ debuild
 dpkg-buildpackage -us -uc -ui -i
dpkg-buildpackage: info: source package debhello
dpkg-buildpackage: info: source version 0.2-1
dpkg-buildpackage: info: source distribution unstable
dpkg-buildpackage: info: source changed by Osamu Aoki <osamu@debian.org>
 dpkg-source -i --before-build .
dpkg-buildpackage: info: host architecture amd64
 debian/rules clean
dh clean
   dh_clean
        rm -f debian/debhelper-build-stamp
 ...
 debian/rules binary
dh binary
   dh_update_autotools_config
   dh_autoreconf
   create-stamp debian/debhelper-build-stamp
   dh_prep
        rm -f -- debian/debhello.substvars
        rm -fr -- debian/.debhelper/generated/debhello/ debian/debhello/ debi...
   dh_auto_install --destdir=debian/debhello/
 ...
Finished running lintian.
```

Let's inspect the result.

The generated files of debhello version 0.2 by the debuild command:

```
[debhello-0.2] $ cd ..
[base_dir] $ tree -FL 1
./
+-- debhello-0.2/
+-- debhello-0.2.tar.xz
+-- 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.xz -> debhello-0.2.tar.xz

2 directories, 8 files
```

You see all the generated files.

- The **debhello_0.2.orig.tar.xz** 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:

```
[base_dir] $ tar --xz -tf debhello-0.2.tar.xz
debhello-0.2/
debhello-0.2/data/
debhello-0.2/data/hello.desktop
debhello-0.2/data/hello.png
debhello-0.2/man/
debhello-0.2/man/hello.1
debhello-0.2/scripts/
debhello-0.2/scripts/hello
debhello-0.2/README.md
[base_dir] $ tar --xz -tf debhello_0.2-1.debian.tar.xz
debian/
debian/README.Debian
debian/changelog
debian/control
debian/copyright
debian/docs
debian/examples
debian/gbp.conf
debian/install
debian/manpages
debian/rules
debian/salsa-ci.yml
debian/source/
debian/source/format
debian/tests/
debian/tests/control
debian/upstream/
debian/upstream/metadata
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:

```
[base_dir] $ 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:

```
[debhello-0.2] $ dpkg -f debhello_0.2-1_all.deb pre-depends \
            depends recommends conflicts breaks
```

(No extra dependency packages required since this is a POSIX shell program.)

注意



If you wish to replace upstream provided PNG file **data/hello.png** with maintainer provided one **debian/hello.png**, editing **debian/install** isn't enough. When you add **debian/hello.png**, you need to add a line 「include-binaries」 to **debian/source/options** since PNG is a binary file. See **dpkg-source**(1).

/tep200.slog/ vim:set filetype=asciidoc:

14.3 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.xz**.

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

```
[base_dir] $ tar --xz -xmf debhello-1.0.tar.xz
[base_dir] $ cd debhello-1.0
[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.xz

```
[base_dir] $ wget http://www.example.org/download/debhello-1.0.tar.xz
 ...
[base_dir] $ tar --xz -xmf debhello-1.0.tar.xz
[base_dir] $ tree
.
+-- debhello-1.0
|   +-- Makefile
|   +-- README.md
|   +-- data
|   |   +-- hello.desktop
|   |   +-- hello.png
|   +-- man
|   |   +-- hello.1
|   +-- scripts
|       +-- hello
+-- debhello-1.0.tar.xz
```

```
5 directories, 7 files
```

Here, the **Makefile** uses **$(DESTDIR)** and **$(prefix)** properly. All other files are the same as in 「項 14.2」and most of the packaging activities are the same.

Makefile (v=1.0)

```
[base_dir] $ 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.

```
[base_dir] $ cd debhello-1.0
[debhello-1.0] $ debmake -b':sh' -x1
I: debmake (version: 5.1.5)
I: Copyright © 2014-2026 Osamu Aoki <osamu@debian.org>
I: [debhello-1.0] $ cd ..
I: Non-native Debian package pkg="debhello", ver="1.0", rev="1" method="dir_d...
I: already in the package-version form: "debhello-1.0"
I: [base_dir] $ ln -sf 'debhello-1.0.tar.xz' 'debhello_1.0.orig.tar.xz'
I: [base_dir] $ cd debhello-1.0
I: parsing option -b ":sh"
I: binary package=debhello Type=script / Arch=all M-A=foreign
I: build_type = make
I: ext_type = 1                       1 files
I: ext_type = desktop                 1 files
I: ext_type = md                      1 files
I: creating debian/* files with "-x 1" option
I: [debhello-1.0] $ licensecheck --recursive --copyright --deb-machine  . > d...
I: creating debian/copyright by licensecheck.
I: creating debian/control from control.py
I: creating debian/control by control.py
I: creating debian/changelog from extra0_changelog
I: creating debian/rules from extra0_rules
I: creating debian/source/format from extra0source_format
 ...
```

Let's inspect the notable template files generated.

debian/rules (template file, v=1.0):

```
[base_dir] $ cd debhello-1.0
[debhello-1.0] $ cat debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
# See debhelper(7) (un-comment to enable)
# This is an autogenerated template for debian/rules.
#
# Output every command that modifies files on the build system.
#export DH_VERBOSE = 1
#
# Copy some variable definitions from pkg-info.mk and vendor.mk
# under /usr/share/dpkg/ to here if they are useful.
#
# These are rarely used code. (START)
#
# The following include for *.mk magically sets miscellaneous
# variables while honoring existing values of pertinent
# environment variables:
#
# Architecture-related variables such as DEB_TARGET_MULTIARCH:
#include /usr/share/dpkg/architecture.mk
# Vendor-related variables such as DEB_VENDOR:
#include /usr/share/dpkg/vendor.mk
# Package-related variables such as DEB_DISTRIBUTION
#include /usr/share/dpkg/pkg-info.mk
#
# You may alternatively set them susing a simple script such as:
# DEB_VENDOR ?= $(shell dpkg-vendor --query  Vendor)
#
# These are rarely used code. (END)
#

### main packaging script based on post dh7 syntax
%:
        dh $@

# debmake generated override targets
# Use "make prefix=/usr" (override prefix=/usr/local in Makefile)
#override_dh_auto_install:
#       dh_auto_install -- prefix=/usr

# Do not install python .pyc .pyo if they exist
#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):

```
[base_dir] $ cd debhello-1.0
[debhello-1.0] $ vim debian/rules
 ... hack, hack, hack, ...
[debhello-1.0] $ cat 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 「項 14.2」.

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

Template files under debian/. (v=1.0):

```
[debhello-1.0] $ rm -f debian/clean debian/dirs debian/install debian/links
[debhello-1.0] $ rm -f debian/README.source debian/source/*.ex
[debhello-1.0] $ rm -rf debian/patches
[debhello-1.0] $ tree -F debian
debian/
+-- README.Debian
+-- changelog
+-- control
+-- copyright
+-- docs
+-- examples
+-- gbp.conf
+-- manpages
+-- rules*
+-- salsa-ci.yml
+-- source/
|   +-- format
+-- tests/
|   +-- control
+-- upstream/
|   +-- metadata
+-- watch

4 directories, 14 files
```

The rest of the packaging activities are practically the same as the ones in 「項 14.2」.

14.4 pyproject.toml (Python3, CLI)

Here is an example of creating a simple Debian package from a Python3 CLI program using **pyproject.toml**.

Let's get the source and make the Debian package.

Download debhello-1.1.tar.xz

```
[base_dir] $ wget http://www.example.org/download/debhello-1.1.tar.xz
 ...
[base_dir] $ tar --xz -xmf debhello-1.1.tar.xz
[base_dir] $ tree
.
+-- debhello-1.1
|   +-- LICENSE
|   +-- MANIFEST.in
|   +-- README.md
|   +-- data
|   |   +-- hello.desktop
|   |   +-- hello.png
|   +-- manpages
|   |   +-- hello.1
|   +-- pyproject.toml
|   +-- src
|       +-- debhello
|           +-- __init__.py
|           +-- main.py
+-- debhello-1.1.tar.xz

6 directories, 10 files
```

Here, the content of this **debhello** source tree as follows.

pyproject.toml (v=1.1) — PEP 517 configuration

```
[base_dir] $ cat debhello-1.1/pyproject.toml
[build-system]
requires = ["setuptools >= 61.0"]  # REQUIRED if [build-system] table is used...
build-backend = "setuptools.build_meta"  # If not defined, then legacy behavi...

[project]
name = "debhello"
version = "1.1.0"
description = "Hello Python (CLI)"
readme = {file = "README.md", content-type = "text/markdown"}
requires-python = ">=3.12"
license = "MIT"
keywords = ["debhello"]
authors = [
  {name = "Osamu Aoki", email = "osamu@debian.org" },
]
maintainers = [
  {name = "Osamu Aoki", email = "osamu@debian.org" },
]
classifiers = [
  "Development Status :: 5 - Production/Stable",
  "Intended Audience :: Developers",
        "Topic :: System :: Archiving :: Packaging",
  "Programming Language :: Python :: 3",
  "Programming Language :: Python :: 3.12",
  "Programming Language :: Python :: 3 :: Only",
  # Others
  "Operating System :: POSIX :: Linux",
  "Natural Language :: English",
]
[project.urls]
"Homepage" = "https://salsa.debian.org/debian/debmake"
"Bug Reports" = "https://salsa.debian.org/debian/debmake/issues"
"Source" = "https://salsa.debian.org/debian/debmake"
[project.scripts]
hello = "debhello.main:main"
[tool.setuptools]
package-dir = {"" = "src"}
packages = ["debhello"]
include-package-data = true
```

MANIFEST.in (v=1.1) — for tar-ball.

```
[base_dir] $ cat debhello-1.1/MANIFEST.in
include data/*
include manpages/*
```

src/debhello/__init__.py (v=1.1)

```
[base_dir] $ cat debhello-1.1/src/debhello/__init__.py
"""
debhello program (CLI)
"""
```

src/debhello/main.py (v=1.1) — command entry point

```
[base_dir] $ cat debhello-1.1/src/debhello/main.py
"""
debhello program
"""

import sys

__version__ = '1.1.0'
```

```
def main():  # needed for console script
    print(' ========== Hello Python3 ==========')
    print('argv = {}'.format(sys.argv))
    print('version = {}'.format(debhello.__version__))
    return

if __name__ == "__main__":
    sys.exit(main())
```

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.

```
[base_dir] $ cd debhello-1.1
[debhello-1.1] $ debmake -b':py3' -x1
I: debmake (version: 5.1.5)
I: Copyright © 2014-2026 Osamu Aoki <osamu@debian.org>
I: [debhello-1.1] $ cd ..
I: Non-native Debian package pkg="debhello", ver="1.1", rev="1" method="dir_d...
I: already in the package-version form: "debhello-1.1"
I: [base_dir] $ ln -sf 'debhello-1.1.tar.xz' 'debhello_1.1.orig.tar.xz'
I: [base_dir] $ cd debhello-1.1
I: parsing option -b ":py3"
I: binary package=debhello Type=python3 / Arch=all M-A=foreign
W: setuptools build system.
I: build_type = Python (pyproject.toml: PEP-518, PEP-621, PEP-660)
I: ext_type = python3                2 files
I: ext_type = 1                      1 files
I: ext_type = desktop                1 files
I: creating debian/* files with "-x 1" option
I: [debhello-1.1] $ licensecheck --recursive --copyright --deb-machine  . > d...
I: creating debian/copyright by licensecheck.
I: creating debian/control from control.py
I: creating debian/control by control.py
I: creating debian/changelog from extra0_changelog
I: creating debian/rules from extra0_rules
 ...
```

Let's inspect the notable template files generated.

debian/rules (template file, v=1.1):

```
[base_dir] $ cd debhello-1.1
[debhello-1.1] $ cat debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
# See debhelper(7) (un-comment to enable)
# This is an autogenerated template for debian/rules.
#
# Output every command that modifies files on the build system.
#export DH_VERBOSE = 1
#
# Copy some variable definitions from pkg-info.mk and vendor.mk
# under /usr/share/dpkg/ to here if they are useful.
#
# These are rarely used code. (START)
#
# The following include for *.mk magically sets miscellaneous
# variables while honoring existing values of pertinent
# environment variables:
#
# Architecture-related variables such as DEB_TARGET_MULTIARCH:
#include /usr/share/dpkg/architecture.mk
# Vendor-related variables such as DEB_VENDOR:
#include /usr/share/dpkg/vendor.mk
# Package-related variables such as DEB_DISTRIBUTION
#include /usr/share/dpkg/pkg-info.mk
```

```
#
# You may alternatively set them susing a simple script such as:
# DEB_VENDOR ?= $(shell dpkg-vendor --query  Vendor)
#
# These are rarely used code. (END)
#

### main packaging script based on post dh7 syntax
%:
        dh $@ --with python3 --buildsystem=pybuild

# debmake generated override targets
# Too complicated to provide examples here.
#
# Check situation of Python on Debian
#     https://wiki.debian.org/Python
#
#     https://wiki.debian.org/Python/TransitionToDHPython2
#     https://wiki.debian.org/Python/Pybuild
#     https://wiki.debian.org/Python/LibraryStyleGuide
#
# If a module package doesn't use distutils or setuptools but uses flit
# you need flit plugin.  See pybuild(1).
#
# Pure PEP-517 based build with "python3 -m build ..." is supported.
#
# To update the upstream source to support python3, see
#     https://wiki.python.org/moin/Python2orPython3
#     https://wiki.python.org/moin/PortingToPy3k/BilingualQuickRef
```

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):

```
[debhello-1.1] $ cat debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: "Osamu Aoki" <osamu@debian.org>
Build-Depends:
 debhelper-compat (= 13),
 dh-python,
 pybuild-plugin-pyproject,
 python3-all,
 python3-setuptools,
Standards-Version: 4.7.3
Homepage: <insert the upstream URL, if relevant>
Rules-Requires-Root: no
#Vcs-Git: https://salsa.debian.org/debian/debhello.git
#Vcs-Browser: https://salsa.debian.org/debian/<project_site>

Package: debhello
Section: unknown
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.
```

```
 .
 ===== This comes from the unmodified template file =====
 .
 Please edit this template file (debian/control) and other package files
 (debian/*) to make them meet all the requirements of the Debian Policy
 before uploading this package to the Debian archive.
 .
 See
  * https://www.debian.org/doc/manuals/developers-reference/best-pkging-pract...
  * https://www.debian.org/doc/manuals/debmake-doc/ch05.en.html#control
 .
 The synopsis description at the top should be about 60 characters and
 written as a phrase.  No extra capital letters or a final period.  No
 articles b''—b'' "a", "an", or "the".
 .
 The package description for general-purpose applications should be
 written for a less technical user.  This means that we should avoid
 jargon.  GNOME or KDE is fine but GTK+ is probably not.
 .
 Use the canonical forms of words:
  * Use X Window System, X11, or X; not X Windows, X-Windows, or X Window.
  * Use GTK+, not GTK or gtk.
  * Use GNOME, not Gnome.
  * Use PostScript, not Postscript or postscript.
```

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 「第 6 章」.

Let's make this Debian package better as the maintainer.

debian/rules (maintainer version, v=1.1):

```
[base_dir] $ cd debhello-1.1
[debhello-1.1] $ vim debian/rules
 ... hack, hack, hack, ...
[debhello-1.1] $ cat debian/rules
#!/usr/bin/make -f
export PYBUILD_NAME=debhello
export PYBUILD_VERBOSE=1
export DH_VERBOSE=1

%:
        dh $@ --with python3 --buildsystem=pybuild
```

debian/control (maintainer version, v=1.1):

```
[debhello-1.1] $ vim debian/control
 ... hack, hack, hack, ...
[debhello-1.1] $ cat debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends:
 debhelper-compat (= 13),
 pybuild-plugin-pyproject,
 python3-all,
Standards-Version: 4.7.3
Rules-Requires-Root: no
Vcs-Browser: https://salsa.debian.org/debian/debmake-doc
Vcs-Git: https://salsa.debian.org/debian/debmake-doc.git
Homepage: https://salsa.debian.org/debian/debmake-doc

Package: debhello
Architecture: all
Depends:
```

```
 ${misc:Depends},
 ${python3:Depends},
Description: Simple packaging example for debmake
 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.

This **debhello** command comes with the upstream-provided manpage and desktop file but the upstream **pyproject.toml** doesn't install them. So you need to update **debian/install** and **debian/manpages** as follows:

debian/install (maintainer version, v=1.1):

```
[debhello-1.1] $ vim debian/copyright
 ... hack, hack, hack, ...
[debhello-1.1] $ cat debian/copyright
Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: debhello
Upstream-Contact: Osamu Aoki <osamu@debian.org>
Source: https://salsa.debian.org/debian/debmake-doc

Files:     *
Copyright: 2015-2024 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.
```

debian/manpages (maintainer version, v=1.1):

```
[debhello-1.1] $ vim debian/install
 ... hack, hack, hack, ...
[debhello-1.1] $ cat debian/install
data/hello.desktop usr/share/applications
data/hello.png usr/share/pixmaps
```

The rest of the packaging activities are practically the same as the ones in 「項 14.3」.

Template files under debian/. (v=1.1):

```
[debhello-1.1] $ rm -f debian/clean debian/dirs debian/links
[debhello-1.1] $ rm -f debian/README.source debian/source/*.ex
[debhello-1.1] $ rm -rf debian/patches
[debhello-1.1] $ tree -F debian
debian/
+-- README.Debian
+-- changelog
+-- control
+-- copyright
+-- docs
```

```
+-- examples
+-- gbp.conf
+-- install
+-- manpages
+-- rules*
+-- salsa-ci.yml
+-- source/
|   +-- format
+-- tests/
|   +-- control
+-- upstream/
|   +-- metadata
+-- watch

4 directories, 15 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:

```
[debhello-1.1] $ dpkg -f debhello_1.1-1_all.deb pre-depends \
           depends recommends conflicts breaks
Depends: python3:any
```

14.5 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 「項 14.3」with enhanced GUI support.
Let's assume its upstream tarball to be **debhello-1.2.tar.xz**.
Let's get the source and make the Debian package.
Download debhello-1.2.tar.xz

```
[base_dir] $ wget http://www.example.org/download/debhello-1.2.tar.xz
 ...
[base_dir] $ tar --xz -xmf debhello-1.2.tar.xz
[base_dir] $ tree
.
+-- debhello-1.2
|   +-- Makefile
|   +-- README.md
|   +-- data
|   |   +-- hello.desktop
|   |   +-- hello.png
|   +-- man
|   |   +-- hello.1
|   +-- scripts
|       +-- hello
+-- debhello-1.2.tar.xz

5 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)

```
[base_dir] $ 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)

```
[base_dir] $ 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 「項 14.3」.

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.

```
[base_dir] $ cd debhello-1.2
[debhello-1.2] $ debmake -b':sh' -x1
I: debmake (version: 5.1.5)
I: Copyright © 2014-2026 Osamu Aoki <osamu@debian.org>
I: [debhello-1.2] $ cd ..
I: Non-native Debian package pkg="debhello", ver="1.2", rev="1" method="dir_d...
I: already in the package-version form: "debhello-1.2"
I: [base_dir] $ ln -sf 'debhello-1.2.tar.xz' 'debhello_1.2.orig.tar.xz'
I: [base_dir] $ cd debhello-1.2
I: parsing option -b ":sh"
I: binary package=debhello Type=script / Arch=all M-A=foreign
I: build_type = make
I: ext_type = 1                       1 files
I: ext_type = desktop                 1 files
I: ext_type = md                      1 files
I: creating debian/* files with "-x 1" option
I: [debhello-1.2] $ licensecheck --recursive --copyright --deb-machine  . > d...
I: creating debian/copyright by licensecheck.
I: creating debian/control from control.py
I: creating debian/control by control.py
I: creating debian/changelog from extra0_changelog
I: creating debian/rules from extra0_rules
I: creating debian/source/format from extra0source_format
 ...
```

Let's inspect the notable template files generated.

debian/control (template file, v=1.2):

```
[debhello-1.2] $ cat debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: "Osamu Aoki" <osamu@debian.org>
Build-Depends:
 debhelper-compat (= 13),
Standards-Version: 4.7.3
Homepage: <insert the upstream URL, if relevant>
Rules-Requires-Root: no
#Vcs-Git: https://salsa.debian.org/debian/debhello.git
#Vcs-Browser: https://salsa.debian.org/debian/<project_site>

Package: debhello
Section: unknown
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.
```

```
 .
 ===== This comes from the unmodified template file =====
 .
 Please edit this template file (debian/control) and other package files
 (debian/*) to make them meet all the requirements of the Debian Policy
 before uploading this package to the Debian archive.
 .
 See
  * https://www.debian.org/doc/manuals/developers-reference/best-pkging-pract...
  * https://www.debian.org/doc/manuals/debmake-doc/ch05.en.html#control
 .
 The synopsis description at the top should be about 60 characters and
 written as a phrase.  No extra capital letters or a final period.  No
 articles b''—b'' "a", "an", or "the".
 .
 The package description for general-purpose applications should be
 written for a less technical user.  This means that we should avoid
 jargon.  GNOME or KDE is fine but GTK+ is probably not.
 .
 Use the canonical forms of words:
  * Use X Window System, X11, or X; not X Windows, X-Windows, or X Window.
  * Use GTK+, not GTK or gtk.
  * Use GNOME, not Gnome.
  * Use PostScript, not Postscript or postscript.
```

Let's make this Debian package better as the maintainer.
debian/control (maintainer version, v=1.2):

```
[debhello-1.2] $ vim debian/control
 ... hack, hack, hack, ...
[debhello-1.2] $ cat debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends:
 debhelper-compat (= 13),
Standards-Version: 4.7.3
Homepage: https://salsa.debian.org/debian/debmake-doc
Rules-Requires-Root: no

Package: debhello
Architecture: all
Multi-Arch: foreign
Depends:
 zenity,
 ${misc:Depends},
Description: Simple packaging example for debmake
 This Debian binary package is an example package.
 (This is an example only)
```

Please note the manually added **zenity** dependency.
The **debian/rules** file is exactly the same as the one in 「項 14.3」.
There are several other template files under the **debian/** directory. These also need to be updated.
Template files under debian/. (v=1.2):

```
[debhello-1.2] $ rm -f debian/clean debian/dirs debian/install debian/links
[debhello-1.2] $ rm -f debian/README.source debian/source/*.ex
[debhello-1.2] $ rm -rf debian/patches
[debhello-1.2] $ tree -F debian
debian/
+-- README.Debian
+-- changelog
+-- control
+-- copyright
```

```
+-- docs
+-- examples
+-- gbp.conf
+-- manpages
+-- rules*
+-- salsa-ci.yml
+-- source/
|   +-- format
+-- tests/
|   +-- control
+-- upstream/
|   +-- metadata
+-- watch

4 directories, 14 files
```

The rest of the packaging activities are practically the same as in 「項 14.3」.
Here is the generated dependency list of **debhello_1.2-1_all.deb**.
The generated dependency list of debhello_1.2-1_all.deb:

```
[debhello-1.2] $ dpkg -f debhello_1.2-1_all.deb pre-depends \
           depends recommends conflicts breaks
Depends: zenity
```

14.6 pyproject.toml (Python3, GUI)

Here is an example of creating a simple Debian package from a Python3 GUI program using **pyproject.toml**.
Let's assume this upstream tarball to be **debhello-1.3.tar.xz**.
Let's get the source and make the Debian package.
Download debhello-1.3.tar.xz

```
[base_dir] $ wget http://www.example.org/download/debhello-1.3.tar.xz
 ...
[base_dir] $ tar --xz -xmf debhello-1.3.tar.xz
[base_dir] $ tree
.
+-- debhello-1.3
|   +-- LICENSE
|   +-- MANIFEST.in
|   +-- README.md
|   +-- data
|   |   +-- hello.desktop
|   |   +-- hello.png
|   +-- manpages
|   |   +-- hello.1
|   +-- pyproject.toml
|   +-- src
|       +-- debhello
|           +-- __init__.py
|           +-- main.py
+-- debhello-1.3.tar.xz

6 directories, 10 files
```

Here, the content of this **debhello** source tree as follows.
pyproject.toml (v=1.3) — PEP 517 configuration

```
[base_dir] $ cat debhello-1.3/pyproject.toml
[build-system]
requires = ["setuptools >= 61.0"]  # REQUIRED if [build-system] table is used...
build-backend = "setuptools.build_meta"  # If not defined, then legacy behavi...
```

```
[project]
name = "debhello"
version = "1.3.0"
description = "Hello Python (GUI)"
readme = {file = "README.md", content-type = "text/markdown"}
requires-python = ">=3.12"
license = "MIT"
keywords = ["debhello"]
authors = [
  {name = "Osamu Aoki", email = "osamu@debian.org" },
]
maintainers = [
  {name = "Osamu Aoki", email = "osamu@debian.org" },
]
classifiers = [
  "Development Status :: 5 - Production/Stable",
  "Intended Audience :: Developers",
        "Topic :: System :: Archiving :: Packaging",
  "Programming Language :: Python :: 3",
  "Programming Language :: Python :: 3.12",
  "Programming Language :: Python :: 3 :: Only",
  # Others
  "Operating System :: POSIX :: Linux",
  "Natural Language :: English",
]
[project.urls]
"Homepage" = "https://salsa.debian.org/debian/debmake"
"Bug Reports" = "https://salsa.debian.org/debian/debmake/issues"
"Source" = "https://salsa.debian.org/debian/debmake"
[project.scripts]
hello = "debhello.main:main"
[tool.setuptools]
package-dir = {"" = "src"}
packages = ["debhello"]
include-package-data = true
```

MANIFEST.in (v=1.3) — for tar-ball.

```
[base_dir] $ cat debhello-1.3/MANIFEST.in
include data/*
include manpages/*
```

src/debhello/__init__.py (v=1.3)

```
[base_dir] $ cat debhello-1.3/src/debhello/__init__.py
"""
debhello program (GUI)
"""
```

src/debhello/main.py (v=1.3) — command entry point

```
[base_dir] $ cat debhello-1.3/src/debhello/main.py
#!/usr/bin/python3
from gi.repository import Gtk

__version__ = '1.3.0'

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()
```

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.

```
[base_dir] $ cd debhello-1.3
[debhello-1.3] $ debmake -b':py3' -x1
I: debmake (version: 5.1.5)
I: Copyright © 2014-2026 Osamu Aoki <osamu@debian.org>
I: [debhello-1.3] $ cd ..
I: Non-native Debian package pkg="debhello", ver="1.3", rev="1" method="dir_d...
I: already in the package-version form: "debhello-1.3"
I: [base_dir] $ ln -sf 'debhello-1.3.tar.xz' 'debhello_1.3.orig.tar.xz'
I: [base_dir] $ cd debhello-1.3
I: parsing option -b ":py3"
I: binary package=debhello Type=python3 / Arch=all M-A=foreign
W: setuptools build system.
I: build_type = Python (pyproject.toml: PEP-518, PEP-621, PEP-660)
I: ext_type = python3                2 files
I: ext_type = 1                      1 files
I: ext_type = desktop                1 files
I: creating debian/* files with "-x 1" option
I: [debhello-1.3] $ licensecheck --recursive --copyright --deb-machine  . > d...
I: creating debian/copyright by licensecheck.
I: creating debian/control from control.py
I: creating debian/control by control.py
I: creating debian/changelog from extra0_changelog
I: creating debian/rules from extra0_rules
 ...
```

The result is practically the same as in 「項 14.4」.

Let's make this Debian package better as the maintainer.

debian/rules (maintainer version, v=1.3):

```
[base_dir] $ cd debhello-1.3
[debhello-1.3] $ vim debian/rules
 ... hack, hack, hack, ...
[debhello-1.3] $ cat debian/rules
#!/usr/bin/make -f
export PYBUILD_NAME=debhello
export PYBUILD_VERBOSE=1
export DH_VERBOSE=1

%:
        dh $@ --with python3 --buildsystem=pybuild
```

debian/control (maintainer version, v=1.3):

```
[debhello-1.3] $ vim debian/control
 ... hack, hack, hack, ...
[debhello-1.3] $ cat debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends:
 debhelper-compat (= 13),
 pybuild-plugin-pyproject,
 python3-all,
Standards-Version: 4.7.3
Homepage: https://salsa.debian.org/debian/debmake-doc
Rules-Requires-Root: no

Package: debhello
Architecture: all
Multi-Arch: foreign
Depends:
 gir1.2-gtk-3.0,
 python3-gi,
 ${misc:Depends},
 ${python3:Depends},
Description: Simple packaging example for debmake
 This Debian binary package is an example package.
 (This is an example only)
```

Please note the manually added **python3-gi** and **gir1.2-gtk-3.0** dependencies.

The rest of the packaging activities are practically the same as in <pyproject>>.

Here is the generated dependency list of **debhello_1.3-1_all.deb**.

The generated dependency list of debhello_1.3-1_all.deb:

```
[debhello-1.3] $ dpkg -f debhello_1.3-1_all.deb pre-depends \
            depends recommends conflicts breaks
Depends: gir1.2-gtk-3.0, python3-gi, python3:any
```

14.7 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 「第 5 章」. 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.xz**.

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

```
[base_dir] $ tar --xz -xmf debhello-1.4.tar.xz
[base_dir] $ cd debhello-1.4
[debhello-1.4] $ make
[debhello-1.4] $ 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.xz

```
[base_dir] $ wget http://www.example.org/download/debhello-1.4.tar.xz
 ...
[base_dir] $ tar --xz -xmf debhello-1.4.tar.xz
[base_dir] $ tree
.
```

```
+-- debhello-1.4
|   +-- LICENSE
|   +-- Makefile
|   +-- README.md
|   +-- data
|   |   +-- hello.desktop
|   |   +-- hello.png
|   +-- man
|   |   +-- hello.1
|   +-- src
|       +-- config.h
|       +-- hello.c
+-- debhello-1.4.tar.xz

5 directories, 9 files
```

Here, the contents of this source are as follows.

src/hello.c (v=1.4):

```
[base_dir] $ 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):

```
[base_dir] $ 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
```

Makefile (v=1.4):

```
[base_dir] $ cat debhello-1.4/src/config.h
#define PACKAGE_AUTHOR "Osamu Aoki"
```

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.

```
[base_dir] $ cd debhello-1.4
[debhello-1.4] $ debmake -x1
I: debmake (version: 5.1.5)
I: Copyright © 2014-2026 Osamu Aoki <osamu@debian.org>
I: [debhello-1.4] $ cd ..
I: Non-native Debian package pkg="debhello", ver="1.4", rev="1" method="dir_d...
I: already in the package-version form: "debhello-1.4"
I: [base_dir] $ ln -sf 'debhello-1.4.tar.xz' 'debhello_1.4.orig.tar.xz'
I: [base_dir] $ cd debhello-1.4
I: parsing option -b ""
I: binary package=debhello Type=bin / Arch=any M-A=foreign
I: build_type = make
I: ext_type = c                       2 files
I: ext_type = 1                       1 files
I: ext_type = desktop                 1 files
I: creating debian/* files with "-x 1" option
I: [debhello-1.4] $ licensecheck --recursive --copyright --deb-machine  . > d...
I: creating debian/copyright by licensecheck.
I: creating debian/control from control.py
I: creating debian/control by control.py
I: creating debian/changelog from extra0_changelog
I: creating debian/rules from extra0_rules
I: creating debian/source/format from extra0source_format
 ...
```

The result is practically the same as in 「項 5.6」.

Let's make this Debian package, which is practically the same as in「項 5.7」, 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 「項 5.7」, 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):

```
[debhello-1.4] $ rm -f debian/clean debian/dirs debian/links
[debhello-1.4] $ rm -f debian/README.source debian/source/*.ex
[debhello-1.4] $ rm -rf debian/patches
[debhello-1.4] $ tree -F debian
debian/
+-- README.Debian
+-- changelog
+-- control
+-- copyright
+-- docs
+-- examples
+-- gbp.conf
+-- install
+-- manpages
+-- rules*
+-- salsa-ci.yml
+-- source/
|   +-- format
+-- tests/
|   +-- control
+-- upstream/
|   +-- metadata
+-- watch
```

```
4 directories, 15 files
```

The rest of the packaging activities are practically the same as the one in 「項 5.8」.

Here is the generated dependency list of all binary packages.

The generated dependency list of all binary packages (v=1.4):

```
[debhello-1.4] $ dpkg -f debhello-dbgsym_1.4-1_amd64.deb pre-depends \
            depends recommends conflicts breaks
Depends: debhello (= 1.4-1)
[debhello-1.4] $ dpkg -f debhello_1.4-1_amd64.deb pre-depends \
            depends recommends conflicts breaks
Depends: libc6 (>= 2.34)
```

14.8 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 「項 14.7」. 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.xz**.

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

```
[base_dir] $ tar --xz -xmf debhello-1.5.tar.xz
[base_dir] $ cd debhello-1.5
[debhello-1.5] $ ./configure --with-math
[debhello-1.5] $ make
[debhello-1.5] $ make install
```

Let's get the source and make the Debian package.

Download debhello-1.5.tar.xz

```
[base_dir] $ wget http://www.example.org/download/debhello-1.5.tar.xz
 ...
[base_dir] $ tar --xz -xmf debhello-1.5.tar.xz
[base_dir] $ tree
.
+-- debhello-1.5
|   +-- LICENSE
|   +-- Makefile.in
|   +-- README.md
|   +-- configure
|   +-- data
|   |   +-- hello.desktop
|   |   +-- hello.png
|   +-- man
|   |   +-- hello.1
|   +-- src
|       +-- hello.c
+-- debhello-1.5.tar.xz

5 directories, 9 files
```

Here, the contents of this source are as follows.

src/hello.c (v=1.5):

```
[base_dir] $ 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):

```
[base_dir] $ 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):

```
[base_dir] $ 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.

```
[base_dir] $ cd debhello-1.5
[debhello-1.5] $ debmake -x1
I: debmake (version: 5.1.5)
I: Copyright © 2014-2026 Osamu Aoki <osamu@debian.org>
I: [debhello-1.5] $ cd ..
I: Non-native Debian package pkg="debhello", ver="1.5", rev="1" method="dir_d...
I: already in the package-version form: "debhello-1.5"
I: [base_dir] $ ln -sf 'debhello-1.5.tar.xz' 'debhello_1.5.orig.tar.xz'
I: [base_dir] $ cd debhello-1.5
I: parsing option -b ""
I: binary package=debhello Type=bin / Arch=any M-A=foreign
I: build_type = configure
I: ext_type = c                        1 files
I: ext_type = 1                        1 files
I: ext_type = desktop                  1 files
I: creating debian/* files with "-x 1" option
I: [debhello-1.5] $ licensecheck --recursive --copyright --deb-machine  . > d...
I: creating debian/copyright by licensecheck.
I: creating debian/control from control.py
I: creating debian/control by control.py
I: creating debian/changelog from extra0_changelog
I: creating debian/rules from extra0_rules
I: creating debian/source/format from extra0source_format
 ...
```

The result is similar to 「項 5.6」but not exactly the same.

Let's inspect the notable template files generated.

debian/rules (template file, v=1.5):

```
[base_dir] $ cd debhello-1.5
[debhello-1.5] $ cat debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
# See debhelper(7) (un-comment to enable)
# This is an autogenerated template for debian/rules.
#
# Output every command that modifies files on the build system.
#export DH_VERBOSE = 1
#
# Copy some variable definitions from pkg-info.mk and vendor.mk
# under /usr/share/dpkg/ to here if they are useful.
#
# See FEATURE AREAS/ENVIRONMENT in dpkg-buildflags(1)
# Apply all hardening options
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
# Package maintainers to append CFLAGS
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
# Package maintainers to append LDFLAGS
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,-O1
#
# With debhelper version 9 or newer, the dh command exports
# all buildflags.  So there is no need to include the
# /usr/share/dpkg/buildflags.mk file here if compat is 9 or newer.
#
# These are rarely used code. (START)
#
# The following include for *.mk magically sets miscellaneous
# variables while honoring existing values of pertinent
# environment variables:
#
# Architecture-related variables such as DEB_TARGET_MULTIARCH:
#include /usr/share/dpkg/architecture.mk
# Vendor-related variables such as DEB_VENDOR:
#include /usr/share/dpkg/vendor.mk
# Package-related variables such as DEB_DISTRIBUTION
#include /usr/share/dpkg/pkg-info.mk
#
# You may alternatively set them susing a simple script such as:
# DEB_VENDOR ?= $(shell dpkg-vendor --query  Vendor)
#
# These are rarely used code. (END)
#

### main packaging script based on post dh7 syntax
%:
        dh $@

# debmake generated override targets
# Multiarch package requires library files to be installed to
# /usr/lib/<triplet>/ .  If the build system does not support
# $(DEB_HOST_MULTIARCH), you may need to override some targets such as
# dh_auto_configure or dh_auto_install to use $(DEB_HOST_MULTIARCH) .
```

Let's make this Debian package better as the maintainer.
debian/rules (maintainer version, v=1.5):

```
[base_dir] $ cd debhello-1.5
[debhello-1.5] $ vim debian/rules
 ... hack, hack, hack, ...
[debhello-1.5] $ 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 $@

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 「項 5.8」.

14.9 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.

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 「項 14.8」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.xz**.

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

```
[base_dir] $ tar --xz -xmf debhello-1.6.tar.xz
[base_dir] $ cd debhello-1.6
[debhello-1.6] $ autoreconf -ivf # optional
[debhello-1.6] $ ./configure --with-math
[debhello-1.6] $ make
[debhello-1.6] $ make install
```

Let's get the source and make the Debian package.

Download debhello-1.6.tar.xz

```
[base_dir] $ wget http://www.example.org/download/debhello-1.6.tar.xz
 ...
[base_dir] $ tar --xz -xmf debhello-1.6.tar.xz
[base_dir] $ tree
.
+-- debhello-1.6
|   +-- LICENSE
|   +-- Makefile.am
|   +-- README.md
|   +-- configure.ac
|   +-- data
|   |   +-- hello.desktop
|   |   +-- hello.png
|   +-- man
|   |   +-- Makefile.am
|   |   +-- hello.1
|   +-- src
|       +-- Makefile.am
|       +-- hello.c
+-- debhello-1.6.tar.xz

5 directories, 11 files
```

Here, the contents of this source are as follows.

src/hello.c (v=1.6):

```
[base_dir] $ 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):

```
[base_dir] $ cat debhello-1.6/Makefile.am
SUBDIRS = src man
[base_dir] $ cat debhello-1.6/man/Makefile.am
dist_man_MANS = hello.1
[base_dir] $ cat debhello-1.6/src/Makefile.am
bin_PROGRAMS = hello
hello_SOURCES = hello.c
```

configure.ac (v=1.6):

```
[base_dir] $ 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
```

ティップ



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.

```
[base_dir] $ cd debhello-1.6
[debhello-1.6] $ debmake -x1
I: debmake (version: 5.1.5)
I: Copyright © 2014-2026 Osamu Aoki <osamu@debian.org>
I: [debhello-1.6] $ cd ..
I: Non-native Debian package pkg="debhello", ver="1.6", rev="1" method="dir_d...
I: already in the package-version form: "debhello-1.6"
I: [base_dir] $ ln -sf 'debhello-1.6.tar.xz' 'debhello_1.6.orig.tar.xz'
I: [base_dir] $ cd debhello-1.6
I: parsing option -b ""
I: binary package=debhello Type=bin / Arch=any M-A=foreign
I: build_type = Autotools with autoreconf
I: ext_type = am                      3 files
I: ext_type = c                       1 files
I: ext_type = 1                       1 files
I: creating debian/* files with "-x 1" option
I: [debhello-1.6] $ licensecheck --recursive --copyright --deb-machine  . > d...
I: creating debian/copyright by licensecheck.
I: creating debian/control from control.py
I: creating debian/control by control.py
I: creating debian/changelog from extra0_changelog
I: creating debian/rules from extra0_rules
I: creating debian/source/format from extra0source_format
 ...
```

The result is similar to 「項 14.8」 but not exactly the same.

Let's inspect the notable template files generated.

debian/rules (template file, v=1.6):

```
[base_dir] $ cd debhello-1.6
[debhello-1.6] $ cat debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
# See debhelper(7) (un-comment to enable)
# This is an autogenerated template for debian/rules.
#
# Output every command that modifies files on the build system.
#export DH_VERBOSE = 1
#
# Copy some variable definitions from pkg-info.mk and vendor.mk
# under /usr/share/dpkg/ to here if they are useful.
#
# See FEATURE AREAS/ENVIRONMENT in dpkg-buildflags(1)
# Apply all hardening options
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
```

```
# Package maintainers to append CFLAGS
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
# Package maintainers to append LDFLAGS
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,-O1
#
# With debhelper version 9 or newer, the dh command exports
# all buildflags.  So there is no need to include the
# /usr/share/dpkg/buildflags.mk file here if compat is 9 or newer.
#
# These are rarely used code. (START)
#
# The following include for *.mk magically sets miscellaneous
# variables while honoring existing values of pertinent
# environment variables:
#
# Architecture-related variables such as DEB_TARGET_MULTIARCH:
#include /usr/share/dpkg/architecture.mk
# Vendor-related variables such as DEB_VENDOR:
#include /usr/share/dpkg/vendor.mk
# Package-related variables such as DEB_DISTRIBUTION
#include /usr/share/dpkg/pkg-info.mk
#
# You may alternatively set them susing a simple script such as:
# DEB_VENDOR ?= $(shell dpkg-vendor --query  Vendor)
#
# These are rarely used code. (END)
#

### main packaging script based on post dh7 syntax
%:
        dh $@ --with autoreconf

# debmake generated override targets
# Set options for ./configure
#CONFIGURE_FLAGS = <options for ./configure>
#overrride_dh_configure:
#       dh_configure -- $(CONFIGURE_FLAGS)
#
# Do not install libtool archive, python .pyc .pyo
#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):

```
[base_dir] $ cd debhello-1.6
[debhello-1.6] $ vim debian/rules
 ... hack, hack, hack, ...
[debhello-1.6] $ 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 $@ --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 「項 5.8」.

14.10 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.

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.xz**.

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

```
[base_dir] $ tar --xz -xmf debhello-1.7.tar.xz
[base_dir] $ cd debhello-1.7
[debhello-1.7] $ mkdir obj-x86_64-linux-gnu # for out-of-tree build
[debhello-1.7] $ cd obj-x86_64-linux-gnu
[debhello-1.7] $ cmake ..
[debhello-1.7] $ make
[debhello-1.7] $ make install
```

Let's get the source and make the Debian package.

Download debhello-1.7.tar.xz

```
[base_dir] $ wget http://www.example.org/download/debhello-1.7.tar.xz
 ...
[base_dir] $ tar --xz -xmf debhello-1.7.tar.xz
[base_dir] $ tree
.
+-- debhello-1.7
|   +-- CMakeLists.txt
|   +-- LICENSE
|   +-- README.md
|   +-- data
|   |   +-- hello.desktop
|   |   +-- hello.png
|   +-- man
|   |   +-- CMakeLists.txt
|   |   +-- hello.1
|   +-- src
|       +-- CMakeLists.txt
|       +-- config.h.in
|       +-- hello.c
+-- debhello-1.7.tar.xz

5 directories, 11 files
```

Here, the contents of this source are as follows.

src/hello.c (v=1.7):

```
[base_dir] $ 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):

```
[base_dir] $ 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):

```
[base_dir] $ cat debhello-1.7/CMakeLists.txt
cmake_minimum_required(VERSION 3.31)
project(debhello)
set(PACKAGE_AUTHOR "Osamu Aoki")
add_subdirectory(src)
add_subdirectory(man)
[base_dir] $ cat debhello-1.7/man/CMakeLists.txt
install(
  FILES ${CMAKE_CURRENT_SOURCE_DIR}/hello.1
  DESTINATION share/man/man1
)
[base_dir] $ 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.

```
[base_dir] $ cd debhello-1.7
[debhello-1.7] $ debmake -x1
I: debmake (version: 5.1.5)
I: Copyright © 2014-2026 Osamu Aoki <osamu@debian.org>
I: [debhello-1.7] $ cd ..
I: Non-native Debian package pkg="debhello", ver="1.7", rev="1" method="dir_d...
I: already in the package-version form: "debhello-1.7"
I: [base_dir] $ ln -sf 'debhello-1.7.tar.xz' 'debhello_1.7.orig.tar.xz'
I: [base_dir] $ cd debhello-1.7
I: parsing option -b ""
I: binary package=debhello Type=bin / Arch=any M-A=foreign
I: build_type = Cmake
I: ext_type = c                       2 files
I: ext_type = 1                       1 files
I: ext_type = desktop                 1 files
I: creating debian/* files with "-x 1" option
I: [debhello-1.7] $ licensecheck --recursive --copyright --deb-machine  . > d...
I: creating debian/copyright by licensecheck.
I: creating debian/control from control.py
I: creating debian/control by control.py
I: creating debian/changelog from extra0_changelog
I: creating debian/rules from extra0_rules
I: creating debian/source/format from extra0source_format
 ...
```

The result is similar to 「項 14.8」but not exactly the same.

Let's inspect the notable template files generated.
debian/rules (template file, v=1.7):

```
[base_dir] $ cd debhello-1.7
[debhello-1.7] $ cat debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
# See debhelper(7) (un-comment to enable)
# This is an autogenerated template for debian/rules.
#
# Output every command that modifies files on the build system.
#export DH_VERBOSE = 1
#
# Copy some variable definitions from pkg-info.mk and vendor.mk
# under /usr/share/dpkg/ to here if they are useful.
#
# See FEATURE AREAS/ENVIRONMENT in dpkg-buildflags(1)
# Apply all hardening options
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
# Package maintainers to append CFLAGS
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
# Package maintainers to append LDFLAGS
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,-O1
#
# With debhelper version 9 or newer, the dh command exports
# all buildflags.  So there is no need to include the
# /usr/share/dpkg/buildflags.mk file here if compat is 9 or newer.
#
# These are rarely used code. (START)
#
# The following include for *.mk magically sets miscellaneous
# variables while honoring existing values of pertinent
# environment variables:
#
# Architecture-related variables such as DEB_TARGET_MULTIARCH:
#include /usr/share/dpkg/architecture.mk
# Vendor-related variables such as DEB_VENDOR:
#include /usr/share/dpkg/vendor.mk
# Package-related variables such as DEB_DISTRIBUTION
#include /usr/share/dpkg/pkg-info.mk
#
# You may alternatively set them susing a simple script such as:
# DEB_VENDOR ?= $(shell dpkg-vendor --query  Vendor)
#
# These are rarely used code. (END)
#

### main packaging script based on post dh7 syntax
%:
        dh $@

# debmake generated override targets
#override_dh_auto_configure:
#       dh_auto_configure -- \
#             -DCMAKE_LIBRARY_ARCHITECTURE="$(DEB_TARGET_MULTIARCH)"
#
# You may need to patch CMakeLists.txt to set the library install path to be:...
#-install(TARGETS <sharedlibname> LIBRARY DESTINATION lib)
#+install(TARGETS <sharedlibname> LIBRARY DESTINATION lib/${CMAKE_LIBRARY_ARC...

# Multiarch package requires library files to be installed to
# /usr/lib/<triplet>/ .  If the build system does not support
# $(DEB_HOST_MULTIARCH), you may need to override some targets such as
# dh_auto_configure or dh_auto_install to use $(DEB_HOST_MULTIARCH) .
```

debian/control (template file, v=1.7):

```
[debhello-1.7] $ cat debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: "Osamu Aoki" <osamu@debian.org>
Build-Depends:
 debhelper-compat (= 13),
 cmake,
Standards-Version: 4.7.3
Homepage: <insert the upstream URL, if relevant>
Rules-Requires-Root: no
#Vcs-Git: https://salsa.debian.org/debian/debhello.git
#Vcs-Browser: https://salsa.debian.org/debian/<project_site>

Package: debhello
Section: unknown
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.
 .
 ===== This comes from the unmodified template file =====
 .
 Please edit this template file (debian/control) and other package files
 (debian/*) to make them meet all the requirements of the Debian Policy
 before uploading this package to the Debian archive.
 .
 See
  * https://www.debian.org/doc/manuals/developers-reference/best-pkging-pract...
  * https://www.debian.org/doc/manuals/debmake-doc/ch05.en.html#control
 .
 The synopsis description at the top should be about 60 characters and
 written as a phrase.  No extra capital letters or a final period.  No
 articles b''—b'' "a", "an", or "the".
 .
 The package description for general-purpose applications should be
 written for a less technical user.  This means that we should avoid
 jargon.  GNOME or KDE is fine but GTK+ is probably not.
 .
 Use the canonical forms of words:
  * Use X Window System, X11, or X; not X Windows, X-Windows, or X Window.
  * Use GTK+, not GTK or gtk.
  * Use GNOME, not Gnome.
  * Use PostScript, not Postscript or postscript.
```

Let's make this Debian package better as the maintainer.

debian/rules (maintainer version, v=1.7):

```
[base_dir] $ cd debhello-1.7
[debhello-1.7] $ vim debian/rules
 ... hack, hack, hack, ...
[debhello-1.7] $ 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 $@

override_dh_auto_configure:
        dh_auto_configure -- -DWITH-MATH=1
```

debian/control (maintainer version, v=1.7):

```
[debhello-1.7] $ vim debian/control
 ... hack, hack, hack, ...
[debhello-1.7] $ cat debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends:
 debhelper-compat (= 13),
 cmake,
Standards-Version: 4.7.3
Homepage: https://salsa.debian.org/debian/debmake-doc
Rules-Requires-Root: no

Package: debhello
Architecture: any
Multi-Arch: foreign
Depends:
 ${misc:Depends},
 ${shlibs:Depends},
Description: Simple packaging example for debmake
 This Debian binary package is an example package.
 (This is an example only)
```

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 「項 14.8」.

14.11 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.

Let's package this in a similar way to 「項 14.9」.

Let's assume this upstream tarball to be **debhello-2.0.tar.xz**.

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

```
[base_dir] $ tar --xz -xmf debhello-2.0.tar.xz
[base_dir] $ cd debhello-2.0
[debhello-2.0] $ autoreconf -ivf # optional
[debhello-2.0] $ ./configure --with-math
[debhello-2.0] $ make
[debhello-2.0] $ make install
```

Let's get the source and make the Debian package.

Download debhello-2.0.tar.xz

```
[base_dir] $ wget http://www.example.org/download/debhello-2.0.tar.xz
 ...
[base_dir] $ tar --xz -xmf debhello-2.0.tar.xz
[base_dir] $ tree
.
+-- debhello-2.0
|   +-- LICENSE
|   +-- Makefile.am
|   +-- README.md
```

```
|   +-- 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.xz

6 directories, 14 files
```

Here, the contents of this source are as follows.

src/hello.c (v=2.0):

```
[base_dir] $ 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):

```
[base_dir] $ cat debhello-2.0/lib/sharedlib.h
int sharedlib();
[base_dir] $ 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):

```
[base_dir] $ cat debhello-2.0/Makefile.am
# recursively process `Makefile.am` in SUBDIRS
SUBDIRS = lib src man
[base_dir] $ cat debhello-2.0/man/Makefile.am
# manpages (distributed in the source package)
dist_man_MANS = hello.1
[base_dir] $ 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
[base_dir] $ 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):

```
[base_dir] $ 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 use the **debmake** command to package this into multiple packages:

- **debhello**: type = **bin**
- **libsharedlib1**: type = **lib**
- **libsharedlib-dev**: type = **dev**

Here, we use the **-b'libsharedlib1,libsharedlib-dev'** option to specify the additional binary packages to be generated.

```
[base_dir] $ cd debhello-2.0
[debhello-2.0] $ debmake -b',libsharedlib1,libsharedlib-dev' -x1
I: debmake (version: 5.1.5)
I: Copyright © 2014-2026 Osamu Aoki <osamu@debian.org>
I: [debhello-2.0] $ cd ..
I: Non-native Debian package pkg="debhello", ver="2.0", rev="1" method="dir_d...
I: already in the package-version form: "debhello-2.0"
I: [base_dir] $ ln -sf 'debhello-2.0.tar.xz' 'debhello_2.0.orig.tar.xz'
I: [base_dir] $ cd debhello-2.0
I: parsing option -b ",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: build_type = Autotools with autoreconf
I: ext_type = am                      4 files
I: ext_type = c                       3 files
I: ext_type = 1                       1 files
I: creating debian/* files with "-x 1" option
I: [debhello-2.0] $ licensecheck --recursive --copyright --deb-machine  . > d...
I: creating debian/copyright by licensecheck.
I: creating debian/control from control.py
I: creating debian/control by control.py
I: creating debian/changelog from extra0_changelog
 ...
```

The result is similar to 「項 14.8」but with more template files.
Let's inspect the notable template files generated.
debian/rules (template file, v=2.0):

```
[base_dir] $ cd debhello-2.0
[debhello-2.0] $ cat debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
# See debhelper(7) (un-comment to enable)
# This is an autogenerated template for debian/rules.
#
# Output every command that modifies files on the build system.
#export DH_VERBOSE = 1
#
# Copy some variable definitions from pkg-info.mk and vendor.mk
# under /usr/share/dpkg/ to here if they are useful.
#
# See FEATURE AREAS/ENVIRONMENT in dpkg-buildflags(1)
# Apply all hardening options
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
# Package maintainers to append CFLAGS
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
# Package maintainers to append LDFLAGS
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,-O1
#
# With debhelper version 9 or newer, the dh command exports
# all buildflags.  So there is no need to include the
# /usr/share/dpkg/buildflags.mk file here if compat is 9 or newer.
#
# These are rarely used code. (START)
#
# The following include for *.mk magically sets miscellaneous
# variables while honoring existing values of pertinent
# environment variables:
#
# Architecture-related variables such as DEB_TARGET_MULTIARCH:
#include /usr/share/dpkg/architecture.mk
# Vendor-related variables such as DEB_VENDOR:
#include /usr/share/dpkg/vendor.mk
```

```
# Package-related variables such as DEB_DISTRIBUTION
#include /usr/share/dpkg/pkg-info.mk
#
# You may alternatively set them susing a simple script such as:
# DEB_VENDOR ?= $(shell dpkg-vendor --query  Vendor)
#
# These are rarely used code. (END)
#

### main packaging script based on post dh7 syntax
%:
        dh $@ --with autoreconf

# debmake generated override targets
# Set options for ./configure
#CONFIGURE_FLAGS = <options for ./configure>
#overrride_dh_configure:
#       dh_configure -- $(CONFIGURE_FLAGS)
#
# Do not install libtool archive, python .pyc .pyo
#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):

```
[base_dir] $ cd debhello-2.0
[debhello-2.0] $ vim debian/rules
 ... hack, hack, hack, ...
[debhello-2.0] $ 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 $@ --with autoreconf

override_dh_missing:
        dh_missing -X.la
```

debian/control (maintainer version, v=2.0):

```
[debhello-2.0] $ vim debian/control
 ... hack, hack, hack, ...
[debhello-2.0] $ cat debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends:
 debhelper-compat (= 13),
 dh-autoreconf,
Standards-Version: 4.7.3
Homepage: https://salsa.debian.org/debian/debmake-doc
Rules-Requires-Root: no

Package: debhello
Architecture: any
Multi-Arch: foreign
Depends:
 libsharedlib1 (= ${binary:Version}),
 ${misc:Depends},
 ${shlibs:Depends},
```

```
Description: Simple packaging example for debmake
 This package contains the compiled binary executable.
 .
 This Debian binary package is an example package.
 (This is an example only)

Package: libsharedlib1
Section: libs
Architecture: any
Multi-Arch: same
Pre-Depends:
 ${misc:Pre-Depends},
Depends:
 ${misc:Depends},
 ${shlibs:Depends},
Description: Simple packaging example for debmake
 This package contains the shared library.

Package: libsharedlib-dev
Section: libdevel
Architecture: any
Multi-Arch: same
Depends:
 libsharedlib1 (= ${binary:Version}),
 ${misc:Depends},
Description: Simple packaging example for debmake
 This package contains the development files.
```

debian/*.install (maintainer version, v=2.0):

```
[debhello-2.0] $ vim debian/copyright
 ... hack, hack, hack, ...
[debhello-2.0] $ cat debian/copyright
Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: debhello
Upstream-Contact: Osamu Aoki <osamu@debian.org>
Source: https://salsa.debian.org/debian/debmake-doc

Files:     *
Copyright: 2015-2021 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.
```

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):

```
[debhello-2.0] $ rm -f debian/clean debian/dirs debian/install debian/links
```

```
[debhello-2.0] $ rm -f debian/README.source debian/source/*.ex
[debhello-2.0] $ rm -rf debian/patches
[debhello-2.0] $ tree -F debian
debian/
+-- README.Debian
+-- changelog
+-- control
+-- copyright
+-- debhello.install
+-- docs
+-- examples
+-- gbp.conf
+-- libsharedlib-dev.install
+-- libsharedlib1.install
+-- libsharedlib1.symbols
+-- manpages
+-- rules*
+-- salsa-ci.yml
+-- source/
|   +-- format
+-- tests/
|   +-- control
+-- upstream/
|   +-- metadata
+-- watch

4 directories, 18 files
```

The rest of the packaging activities are practically the same as the one in 「項 14.8」.

Here are the generated dependency list of all binary packages.

The generated dependency list of all binary packages (v=2.0):

```
[debhello-2.0] $ dpkg -f debhello-dbgsym_2.0-1_amd64.deb pre-depends \
            depends recommends conflicts breaks
Depends: debhello (= 2.0-1)
[debhello-2.0] $ dpkg -f debhello_2.0-1_amd64.deb pre-depends \
            depends recommends conflicts breaks
Depends: libsharedlib1 (= 2.0-1), libc6 (>= 2.34)
[debhello-2.0] $ dpkg -f libsharedlib-dev_2.0-1_amd64.deb pre-depends \
            depends recommends conflicts breaks
Depends: libsharedlib1 (= 2.0-1)
[debhello-2.0] $ dpkg -f libsharedlib1-dbgsym_2.0-1_amd64.deb pre-depends \
            depends recommends conflicts breaks
Depends: libsharedlib1 (= 2.0-1)
[debhello-2.0] $ dpkg -f libsharedlib1_2.0-1_amd64.deb pre-depends \
            depends recommends conflicts breaks
Depends: libc6 (>= 2.2.5)
```

14.12 CMake (multi-binary package)

This example demonstrates 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 files such as **config.h.in**) as its build system.

Let's assume this upstream tarball to be **debhello-2.1.tar.xz**.

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

```
[base_dir] $ tar --xz -xmf debhello-2.1.tar.xz
[base_dir] $ cd debhello-2.1
[debhello-2.1] $ mkdir obj-x86_64-linux-gnu
[debhello-2.1] $ cd obj-x86_64-linux-gnu
[debhello-2.1] $ cmake ..
[debhello-2.1] $ make
```

```
[debhello-2.1] $ make install
```

Let's get the source and make the Debian package.
Download debhello-2.1.tar.xz

```
[base_dir] $ wget http://www.example.org/download/debhello-2.1.tar.xz
 ...
[base_dir] $ tar --xz -xmf debhello-2.1.tar.xz
[base_dir] $ tree
.
+-- debhello-2.1
|   +-- CMakeLists.txt
|   +-- LICENSE
|   +-- README.md
|   +-- 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.xz

6 directories, 14 files
```

Here, the contents of this source are as follows.
src/hello.c (v=2.1):

```
[base_dir] $ 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):

```
[base_dir] $ 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):

```
[base_dir] $ cat debhello-2.1/lib/sharedlib.h
int sharedlib();
[base_dir] $ 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):

```
[base_dir] $ cat debhello-2.1/CMakeLists.txt
cmake_minimum_required(VERSION 3.31)
project(debhello)
set(PACKAGE_AUTHOR "Osamu Aoki")
add_subdirectory(lib)
add_subdirectory(src)
add_subdirectory(man)
[base_dir] $ cat debhello-2.1/man/CMakeLists.txt
install(
  FILES ${CMAKE_CURRENT_SOURCE_DIR}/hello.1
  DESTINATION share/man/man1
)
[base_dir] $ 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.

```
[base_dir] $ cd debhello-2.1
[debhello-2.1] $ debmake -b',libsharedlib1,libsharedlib-dev' -x1
I: debmake (version: 5.1.5)
I: Copyright © 2014-2026 Osamu Aoki <osamu@debian.org>
I: [debhello-2.1] $ cd ..
I: Non-native Debian package pkg="debhello", ver="2.1", rev="1" method="dir_d...
I: already in the package-version form: "debhello-2.1"
I: [base_dir] $ ln -sf 'debhello-2.1.tar.xz' 'debhello_2.1.orig.tar.xz'
I: [base_dir] $ cd debhello-2.1
I: parsing option -b ",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: build_type = Cmake
I: ext_type = c                        4 files
I: ext_type = 1                        1 files
I: ext_type = desktop                  1 files
I: creating debian/* files with "-x 1" option
I: [debhello-2.1] $ licensecheck --recursive --copyright --deb-machine  . > d...
I: creating debian/copyright by licensecheck.
I: creating debian/control from control.py
I: creating debian/control by control.py
I: creating debian/changelog from extra0_changelog
 ...
```

The result is similar to 「項 14.8」 but not exactly the same.
Let's inspect the notable template files generated.
debian/rules (template file, v=2.1):

```
[base_dir] $ cd debhello-2.1
[debhello-2.1] $ cat debian/rules
#!/usr/bin/make -f
```

```
# You must remove unused comment lines for the released package.
# See debhelper(7) (un-comment to enable)
# This is an autogenerated template for debian/rules.
#
# Output every command that modifies files on the build system.
#export DH_VERBOSE = 1
#
# Copy some variable definitions from pkg-info.mk and vendor.mk
# under /usr/share/dpkg/ to here if they are useful.
#
# See FEATURE AREAS/ENVIRONMENT in dpkg-buildflags(1)
# Apply all hardening options
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
# Package maintainers to append CFLAGS
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
# Package maintainers to append LDFLAGS
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,-O1
#
# With debhelper version 9 or newer, the dh command exports
# all buildflags.  So there is no need to include the
# /usr/share/dpkg/buildflags.mk file here if compat is 9 or newer.
#
# These are rarely used code. (START)
#
# The following include for *.mk magically sets miscellaneous
# variables while honoring existing values of pertinent
# environment variables:
#
# Architecture-related variables such as DEB_TARGET_MULTIARCH:
#include /usr/share/dpkg/architecture.mk
# Vendor-related variables such as DEB_VENDOR:
#include /usr/share/dpkg/vendor.mk
# Package-related variables such as DEB_DISTRIBUTION
#include /usr/share/dpkg/pkg-info.mk
#
# You may alternatively set them susing a simple script such as:
# DEB_VENDOR ?= $(shell dpkg-vendor --query  Vendor)
#
# These are rarely used code. (END)
#

### main packaging script based on post dh7 syntax
%:
        dh $@

# debmake generated override targets
#override_dh_auto_configure:
#       dh_auto_configure -- \
#             -DCMAKE_LIBRARY_ARCHITECTURE="$(DEB_TARGET_MULTIARCH)"
#
# You may need to patch CMakeLists.txt to set the library install path to be:...
#-install(TARGETS <sharedlibname> LIBRARY DESTINATION lib)
#+install(TARGETS <sharedlibname> LIBRARY DESTINATION lib/${CMAKE_LIBRARY_ARC...

# Multiarch package requires library files to be installed to
# /usr/lib/<triplet>/ .  If the build system does not support
# $(DEB_HOST_MULTIARCH), you may need to override some targets such as
# dh_auto_configure or dh_auto_install to use $(DEB_HOST_MULTIARCH) .
```

Let's make this Debian package better as the maintainer.

debian/rules (maintainer version, v=2.1):

```
[base_dir] $ cd debhello-2.1
[debhello-2.1] $ vim debian/rules
 ... hack, hack, hack, ...
```

```
[debhello-2.1] $ 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
DEB_HOST_MULTIARCH ?= $(shell dpkg-architecture -qDEB_HOST_MULTIARCH)

%:
        dh $@

override_dh_auto_configure:
        dh_auto_configure -- \
              -DCMAKE_LIBRARY_ARCHITECTURE="$(DEB_HOST_MULTIARCH)"
```

debian/control (maintainer version, v=2.1):

```
[debhello-2.1] $ vim debian/control
 ... hack, hack, hack, ...
[debhello-2.1] $ cat debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends:
 debhelper-compat (= 13),
 cmake,
Standards-Version: 4.7.3
Homepage: https://salsa.debian.org/debian/debmake-doc
Rules-Requires-Root: no

Package: debhello
Architecture: any
Multi-Arch: foreign
Depends:
 libsharedlib1 (= ${binary:Version}),
 ${misc:Depends},
 ${shlibs:Depends},
Description: Simple packaging example for debmake
 This package contains the compiled binary executable.
 .
 This Debian binary package is an example package.
 (This is an example only)

Package: libsharedlib1
Section: libs
Architecture: any
Multi-Arch: same
Pre-Depends:
 ${misc:Pre-Depends},
Depends:
 ${misc:Depends},
 ${shlibs:Depends},
Description: Simple packaging example for debmake
 This package contains the shared library.

Package: libsharedlib-dev
Section: libdevel
Architecture: any
Multi-Arch: same
Depends:
 libsharedlib1 (= ${binary:Version}),
 ${misc:Depends},
Description: Simple packaging example for debmake
```

```
 This package contains the development files.
```

debian/*.install (maintainer version, v=2.1):

```
[debhello-2.1] $ vim debian/copyright
 ... hack, hack, hack, ...
[debhello-2.1] $ cat debian/copyright
Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: debhello
Upstream-Contact: Osamu Aoki <osamu@debian.org>
Source: https://salsa.debian.org/debian/debmake-doc

Files:     *
Copyright: 2015-2021 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.
```

The upstream CMakeLists.txt file needs to be patched to handle the multiarch path correctly.

debian/patches/* (maintainer version, v=2.1):

```
 ... hack, hack, hack, ...
[debhello-2.1] $ cat 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):

```
[debhello-2.1] $ rm -f debian/clean debian/dirs debian/install debian/links
[debhello-2.1] $ rm -f debian/README.source debian/source/*.ex
[debhello-2.1] $ tree -F debian
debian/
+-- README.Debian
+-- changelog
+-- control
+-- copyright
+-- debhello.install
+-- docs
+-- examples
+-- gbp.conf
+-- libsharedlib-dev.install
+-- libsharedlib1.install
+-- libsharedlib1.symbols
+-- manpages
+-- patches/
|   +-- 000-cmake-multiarch.patch
|   +-- series
```

```
+-- rules*
+-- salsa-ci.yml
+-- source/
|   +-- format
+-- tests/
|   +-- control
+-- upstream/
|   +-- metadata
+-- watch

5 directories, 20 files
```

The rest of the packaging activities are practically the same as the one in 「項 14.8」.

Here are the generated dependency list of all binary packages.

The generated dependency list of all binary packages (v=2.1):

```
[debhello-2.1] $ dpkg -f debhello-dbgsym_2.1-1_amd64.deb pre-depends \
           depends recommends conflicts breaks
Depends: debhello (= 2.1-1)
[debhello-2.1] $ dpkg -f debhello_2.1-1_amd64.deb pre-depends \
           depends recommends conflicts breaks
Depends: libsharedlib1 (= 2.1-1), libc6 (>= 2.34)
[debhello-2.1] $ dpkg -f libsharedlib-dev_2.1-1_amd64.deb pre-depends \
           depends recommends conflicts breaks
Depends: libsharedlib1 (= 2.1-1)
[debhello-2.1] $ dpkg -f libsharedlib1-dbgsym_2.1-1_amd64.deb pre-depends \
           depends recommends conflicts breaks
Depends: libsharedlib1 (= 2.1-1)
[debhello-2.1] $ dpkg -f libsharedlib1_2.1-1_amd64.deb pre-depends \
           depends recommends conflicts breaks
Depends: libc6 (>= 2.2.5)
```

14.13 Internationalization

Here is an example of updating the simple upstream C source **debhello-2.0.tar.xz** presented in 「項 14.11」for internationalization (i18n) and creating the updated upstream C source **debhello-2.0.tar.xz**.

In the real situation, the package should already be internationalized. So this example is educational for you to understand how this internationalization is implemented.

ティップ



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.xz (i18n)

```
[base_dir] $ wget http://www.example.org/download/debhello-2.0.tar.xz
 ...
[base_dir] $ tar --xz -xmf debhello-2.0.tar.xz
[base_dir] $ tree
.
+-- debhello-2.0
|   +-- LICENSE
|   +-- Makefile.am
|   +-- README.md
|   +-- 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.xz

6 directories, 14 files
```

Internationalize this source tree with the **gettextize** command and remove files auto-generated by Autotools.

run gettextize (i18n):

```
[base_dir] $ 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.
[debhello-2.0] $ rm -rf m4 build-aux *~
```

Let's check generated files under the **po/** directory.
files in po (i18n):

```
[debhello-2.0] $ ls -l po
total 60
-rw-rw-r-- 1 osamu osamu   494 Jul 22 15:31 ChangeLog
-rw-rw-r-- 1 osamu osamu 17577 Jul 22 15:31 Makefile.in.in
-rw-rw-r-- 1 osamu osamu  3376 Jul 22 15:31 Makevars.template
-rw-rw-r-- 1 osamu osamu    59 Jul 22 15:31 POTFILES.in
-rw-rw-r-- 1 osamu osamu  2203 Jul 22 15:31 Rules-quot
-rw-rw-r-- 1 osamu osamu   217 Jul 22 15:31 boldquot.sed
-rw-rw-r-- 1 osamu osamu  1337 Jul 22 15:31 en@boldquot.header
-rw-rw-r-- 1 osamu osamu  1203 Jul 22 15:31 en@quot.header
-rw-rw-r-- 1 osamu osamu   672 Jul 22 15:31 insert-header.sin
-rw-rw-r-- 1 osamu osamu   153 Jul 22 15:31 quot.sed
-rw-rw-r-- 1 osamu osamu   432 Jul 22 15:31 remove-potcdate.sin
```

Let's update the **configure.ac** by adding 「**AM_GNU_GETTEXT([external])**」, etc..
configure.ac (i18n):

```
[debhello-2.0] $ vim configure.ac
 ... hack, hack, hack, ...
[debhello-2.0] $ 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, ...
[debhello-2.0] $ diff -u po/Makevars.template po/Makevars
--- po/Makevars.template        2026-07-22 15:31:09.890068240 +0900
+++ po/Makevars 2026-07-22 15:31:09.986012740 +0900
@@ -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:
[debhello-2.0] $ rm po/Makevars.template
```

Let's update C sources for the i18n version by wrapping strings with **_(⋯)**.
src/hello.c (i18n):

```
 ... hack, hack, hack, ...
[debhello-2.0] $ cat src/hello.c
#include "config.h"
#include <stdio.h>
#include <sharedlib.h>
#include <libintl.h>
#define _(string) gettext (string)
int
main()
{
        printf(_("Hello, I am " PACKAGE_AUTHOR "!\n"));
        sharedlib();
        return 0;
}
```

lib/sharedlib.c (i18n):

```
 ... hack, hack, hack, ...
[debhello-2.0] $ cat lib/sharedlib.c
#include <stdio.h>
#include <libintl.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):

```
[debhello-2.0] $ fgrep -v '[ja]=' data/hello.desktop > data/hello.desktop.in
[debhello-2.0] $ rm data/hello.desktop
[debhello-2.0] $ 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, ...
[debhello-2.0] $ 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):

```
[debhello-2.0] $ 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):

```
[debhello-2.0] $ xgettext -f po/POTFILES.in -d debhello -o po/debhello.pot -k...
[debhello-2.0] $ 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: 2026-07-22 15:31+0900\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:9
#, c-format
msgid "Hello, I am "
```

```
msgstr ""

#: lib/sharedlib.c:7
#, c-format
msgid "This is a shared library!\n"
msgstr ""

#: data/hello.desktop.in:2
msgid "Hello"
msgstr ""

#: data/hello.desktop.in:3
msgid "Greetings"
msgstr ""

#: data/hello.desktop.in:5
msgid "hello"
msgstr ""
```

Let's add a translation for French.
po/LINGUAS and po/fr.po (i18n):

```
[debhello-2.0] $ echo 'fr' > po/LINGUAS
[debhello-2.0] $ cp po/debhello.pot po/fr.po
[debhello-2.0] $ vim po/fr.po
 ... hack, hack, hack, ...
[debhello-2.0] $ 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 「項 14.11」.
You can find more i18n examples by following 「項 14.14」.

14.14 Details

You can obtain detailed information about the examples presented and their variants as follows:

How to get details

```
[base_dir] $ apt-get source debmake-doc
[base_dir] $ cd debmake-doc*
[debmake-doc-*] $ view examples/README.md
```

Follow the exact instruction in **examples/README.md**.

```
[debmake-doc-*] $ cd examples
[examples] $ make
```

Now, each directory named as **examples/debhello-?.?_build-?** 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 **package** directory
- snapshot source tree image after the **debuild** command: the **test** directory

Notable examples include:

- POSIX shell script with Makefile and i18n support (v=3.0)
- C source with Makefile.in + configure and i18n support (v=3.2)
- C source with Autotools and i18n support (v=3.3)
- C source with CMake and i18n support (v=3.4)

Chapter 15

debmake(1) manpage

15.1 NAME

debmake - program to make a Debian source package

15.2 SYNOPSIS

debmake [**-h**] [**-n**] [**-p** *package*] [**-u** *version*] [**-r** *revision*] [**-z** *extension*] [**-b** "*binarypackage[:type],* ⋯]" [**-D** *value*] [**-e** *foo@example.org*] [**-f** "*firstname lastname*"] [**-i** [**debuild**|**sbuild**|**dgit sbuild**|**gbp buildpackage**|**dpkg-buildpackage**| ⋯]] [**-m**] [**-q**] [**-v**] [**-V**] [**-w** *"addon,* ⋯*"*] [**-x** [01234]] [**-y**] [**-B**] [*URL*]

15.3 DESCRIPTION

debmake helps to build the Debian package from the upstream source.

Normally, this is done as follows:

- The upstream source is obtained as a tarball from a remote web site or a cloned work tree using 「**git clone**」.
  - For a tarball, it is expanded to many files in the source directory.
  - For a cloned work tree, it is used as the source directory.
- debmake is typically invoked in the source directory without any argument.
  - The source directory is copied to ../*package-version*/ directory.
  - If ../*package_version*.**orig.tar.xz** is missing, it is generated.
  - The current directory is moved to ../package-version/.
  - Template files are generated in the ../*package-version*/**debian/** directory
- Files in the ../*package-version*/**debian/** directory should be manually adjusted.
- **dpkg-buildpackage** (usually from its wrapper **debuild**, **sbuild**, ⋯) is invoked in the ../*package-version*/ directory to make Debian source and binary packages.

Also, **debmake** can be invoked with an argument. This argument can be *URL* for a tarball hosted on a remote web site or for a source code accessed by 「**git clone**」; or local *PATH* to the tarball or the source code.

Arguments to **-b**, **-f**, and **-w** options need to be quoted to protect them from the shell.

Other tools also offer ways to obtain the upstream tarball and creating required symlink to build a Debian package depending on your workflow. For example, **origtargz**, **mk-origtargz**, **git-deborig**, and **pristine-tar**.

15.4 Positional arguments

URL acquire the source tree from the tarball, the git repository or the source tree at this *URL* (or *PATH*) (if missing, the source tree uses the current directory)

15.5 Options

-h, --help show this help message and exit

-n, **--native** make a native source package without **.orig.tar.xz**

-p, --package package set the Debian package name

-u, --upstreamversion *version* set the upstream package version ("@" in *version* is replaced by "**0~yymmddHHMM**" timestamp)

-r, --revision *revision* set the Debian package revision ("@" in *revision* is replaced by "**0~yymmddHHMM**" timestamp)

-z, --tarz *extension* set the tarball compression type for the missing upstream tarball, *extension*=(**tar.xz**|**tar.gz**|**tar.bz2**
(alias: **z**, **b**, **x**)

-b, --binaryspec *"binarypackage[:type],* ⋯*"* set the binary package specs by a comma separated list of *binarypackage:type* pairs. 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**)
- **python3**: Python (version 3) script package (all, foreign) (alias: **py3**, **python**, **py**)
- **ruby**: Ruby script package (all, foreign) (alias: **rb**)
- **nodejs**: Node.js based JavaScript package (all, foreign) (alias: **js**)
- **script**: Shell and other interpreted language 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**.

Here are examples for typical binary package split scenarios where the upstream Debian source package name is **foo**:

- Generating an executable binary package **foo**:
  - 「**-b'foo:bin'**」, or its short form 「**-b'-'**」, or no **-b** option
- Generating an executable (python3) binary package **python3-foo**:
  - 「**-b'python3-foo:py'**」, or its short form 「**-b'python3-foo'**」
- Generating a data package **foo**:
  - 「**-b'foo:data'**」, or its short form 「**-b'-:data'**」
- Generating a executable binary package **foo** and a documentation one **foo-doc**:
  - 「**-b'foo:bin,foo-doc:doc'**」, or its short form 「**-b'-:-doc'**」
- Generating a executable binary package **foo**, a library package **libfoo1**, and a library development package **libfoo-dev**:

– 「**-b'foo:bin,libfoo1:lib,libfoo-dev:dev'**」 or its short form 「**-b'-,libfoo1,libfoo-dev'**」

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

-e, --email *foo@example.org* set e-mail address

The default is taken from the value of the environment variable **$DEBEMAIL**.

-D, --debug *value* set **DEBUG** environment variable to *value* for debug logging (substring of "**spPd**", use "_" to unset **DEBUG**)

-f, --fullname *"firstname lastname"* set the fullname

The default is taken from the value of the environment variable **$DEBFULLNAME**.

-i, --invoke *[debuild|sbuild|dgit sbuild|gbp buildpackage|dpkg-buildpackage|* ⋯*]* invoke package build tool

-m, --monoarch force packages to be non-multiarch

-q, --quitearly quit early before creating files in the debian directory

-v, --version show version information

-V, --verbose use --verbose for shell commands if available

-w, --with *"addon* ⋯*"* set additional 「**dh --with**」 option arguments in **debian/rules**

For Autotools based packages, if they install Python (version 3) programs, setting **python3** as *addon* to the **debmake** command argument is needed since this is non-obvious. But for **pyproject.toml** based Python packages, setting **python3** as *addon* to the **debmake** command argument is not needed since this is obvious and the **debmake** command automatically set it to the **dh**(1) command.

-x, --extra *[01234]* generate extra configuration files as templates (default: 2)

Please note **debian/changelog**, **debian/control**, **debian/copyright**, **debian/rules**, and **debian/source/format** are required configuration files to build a modern Debian binary package.

The number determines which configuration templates are generated.

- **-x0**: all 5 required configuration template files. (selected option if any of these required files already exist)
- **-x1**: all **-x0** files + desirable configuration template files with binary package type supports.
- **-x2**: all **-x1** files + normal configuration template files with maintainer script supports. (default)
- **-x3**: all **-x2** files + optional configuration template files.
- **-x4**: all **-x3** files + deprecated configuration template files.

Some configuration template files are generated with the extra **.ex** suffix to ease their removal. To activate these, rename their file names to the ones without the **.ex** suffix and edit their contents. Existing configuration files are never overwritten. If you wish to update some of the existing configuration files, please rename them before running the **debmake** command and manually merge the generated configuration files with the old renamed ones.

-y, --yes use once to 「force yes」 for all prompts, twice to 「force no」

-B, --backup keep the user edited ones without **.ex** suffix and create template files with **.ex** suffix

15.6 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.

For a typical C program source tree packaged with **autoconf**/**automake**:

- **debmake -i sbuild**

For a typical Python (version 3) module source tree:

- **debmake -b":python3" -i sbuild**

For a typical Python (version 3) module in the *package-version***.tar.xz** archive:

- **debmake** *package-version***.tar.xz -b":python3" -i sbuild**

For a typical Perl module in the *package-version***.tar.xz** archive:

- **debmake** *package-version***.tar.xz -b":perl" -i sbuild**

15.7 HELPER PACKAGES

Packaging may require installation of some additional specialty helper packages.

- Python (version 3) programs may require the **pybuild-plugin-pyproject** package.
- The Autotools (**autoconf** + **automake**) build system may require **autotools-dev** or **dh-autoreconf** package.
- Ruby programs may require the **gem2deb** package.
- Node.js based JavaScript programs may require the **pkg-js-tools** package.
- Java programs may require the **javahelper** package.
- Gnome programs may require the **gobject-introspection** package.
- etc.

15.8 CAVEAT

Although **debmake** is meant to provide template files for the package maintainer to work on, actual packaging activities are often performed without using **debmake** while referencing only existing similar packages and 「Debian Policy Manual」. All template files generated by **debmake** are required to be modified manually.

There are some points for **debmake**:

- **debmake** helps to write terse packaging tutorial 「Guide for Debian Maintainers」 (**debmake-doc** package).
- **debmake** provides short extracted license texts as **debian/copyright** in decent accuracy to help license review.
- 「Guide for Debian Maintainers」 also serves as a tutorial with examples for the usage of debmake.
- **debmake** internally calls **licensecheck** from the **licensecheck** package to create **debian/copyright** if it doesn't exist.
- **debmake** internally calls **lrc** from the **licenserecon** package to verify **debian/copyright** if it already exists.

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.

15.9 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.

- **s**: program progress logging
- **p**: key para[..] value logging
- **P**: all para[..] value logging
- **d**: para["debs"] value logging

Use this feature as:

```
[base_dir] $ export DEBUG=spd; debmake ...
```

or

```
[base_dir] $ debmake -D spd ...
```

See **README.md** in the source for more.

15.10 AUTHOR

Copyright © 2014-2026 Osamu Aoki <osamu@debian.org>

15.11 LICENSE

Expat License

15.12 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 **licensecheck**(1), **lrc**(1), **dpkg-source**(1), **deb-control**(5), **debhelper**(7), **dh**(1), **dpkg-buildpackage**(1), **debuild**(1), **quilt**(1), **dpkg-depcheck**(1), **sbuild**(1), **gbp-buildpackage**(1), and **gbp-pq**(1) manpages.

Chapter 16

debmake options

Here are some additional explanations for **debmake** options.

16.1 Shortcut option (-i)

The **debmake** command offers a shortcut option.

- **-i** : execute script to build the binary package

The example in the above 「第 5 章」can be done simply as follows.

```
[base_dir] $ debmake package-1.0.tar.xz -i debuild
```

ティップ



A URL such as 「**https://www.example.org/DL/package-1.0.tar.xz**」for a tarball, 「**https://github.com/username/package.git**」for a git repository, or 「**/path/to/source_dir**」for a local source tree may be used as an argument.

16.2 debmake -b

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

- **Package:**
- **Architecture:** (e.g. **amd64**)
- **Multi-Arch:** (see 「項 10.10」)
- **Depends:**
- **Pre-Depends:**

The **debmake** command also sets an appropriate set of substvars (substitution variables) used in each pertinent dependency stanza.

Let's quote the pertinent part from the **debmake** manpage here.

-b, --binaryspec ***"binarypackage[:type],*** ⋯***"*** set the binary package specs by a comma separated list of *binarypackage:type* pairs. 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**)
- **python3**: Python (version 3) script package (all, foreign) (alias: **py3**, **python**, **py**)
- **ruby**: Ruby script package (all, foreign) (alias: **rb**)
- **nodejs**: Node.js based JavaScript package (all, foreign) (alias: **js**)
- **script**: Shell and other interpreted language 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**.

Here are examples for typical binary package split scenarios where the upstream Debian source package name is **foo**:

- Generating an executable binary package **foo**:
  - 「**-b'foo:bin'**」, or its short form 「**-b'-'**」, or no **-b** option
- Generating an executable (python3) binary package **python3-foo**:
  - 「**-b'python3-foo:py'**」, or its short form 「**-b'python3-foo'**」
- Generating a data package **foo**:
  - 「**-b'foo:data'**」, or its short form 「**-b'-:data'**」
- Generating a executable binary package **foo** and a documentation one **foo-doc**:
  - 「**-b'foo:bin,foo-doc:doc'**」, or its short form 「**-b'-:-doc'**」
- Generating a executable binary package **foo**, a library package **libfoo1**, and a library development package **libfoo-dev**:
  - 「**-b'foo:bin,libfoo1:lib,libfoo-dev:dev'**」or its short form 「**-b'-,libfoo1,libfoo-dev'**」

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

16.3 debmake -B

The **debmake** command invoked with the **-B** option can generate template files with **.ex** suffix. This is handy if you want to see auto-generated template files to the existing ones.

16.4 debmake -x

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

- See 「項 14.1」for cherry-picking of the template files.

注意



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