



debian

Leitfaden für Debian-Betreuer

Osamu Aoki, Helge Kreutzmann, and Mechtilde Stehmann

August 20, 2021

Leitfaden für Debian-Betreuer

by Osamu Aoki, Helge Kreutzmann, and Mechtilde Stehmann

Copyright © 2014-2020 Osamu Aoki

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

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

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

Diese Anleitung wurde mit den nachfolgenden Dokumenten als Referenz erstellt:

- »Making a Debian Package (AKA the Debmake Manual)«, Copyright © 1997 Jaldhar Vyas.
- »The New-Maintainer's Debian Packaging Howto«, Copyright © 1997 Will Lowe.
- »Debian-Leitfaden für Neue Paketbetreuer«, Copyright © 1998-2002 Josip Rodin, 2005-2017 Osamu Aoki, 2010 Craig Small und 2010 Raphaël Hertzog.

Die neuste Version dieser Anleitung sollte

- im Paket [debmake-doc](#) und
- auf der [Debian-Dokumentations-Website](#) verfügbar sein.

Contents

1	Überblick	1
2	Voraussetzungen	3
2.1	Leute bei Debian	3
2.2	Wie Sie beitragen können	3
2.3	Soziale Dynamik von Debian	4
2.4	Technische Erinnerungen	4
2.5	Debian-Dokumentation	5
2.6	Hilfequellen	5
2.7	Archivsituation	6
2.8	Wege, beizutragen	6
2.9	Neue Beitragende und Betreuer	8
3	Werkzeugeinrichtung	9
3.1	E-Mail-Adresse	9
3.2	MC	9
3.3	Git	10
3.4	Quilt	10
3.5	Devscripts	11
3.6	sbuild	11
3.7	git-buildpackage	12
3.8	HTTP-Proxy	13
3.9	Privates Debian-Depot	13
3.10	Virtual machines	13
3.11	Local network with virtual machines	13
4	Einfaches Beispiel	14
4.1	Packaging tarball	14
4.2	Gesamtbild	14
4.3	Was ist Debmake?	15
4.4	Was ist Debuild?	16
4.5	Schritt 1: Holen der Quellen der Originalautoren	16
4.6	Schritt 2: Mit Debmake erstellte Vorlagendateien	17
4.7	Schritt 3: Anpassung der Vorlagendateien	21
4.8	Schritt 4: Paketbau mit debuild	23
4.9	Schritt 3 (alternativ): Änderungen der Quellen der Originalautoren	26
4.9.1	Patch durch diff -u	26
4.9.2	Patch durch Dquilt	27
4.9.3	Patch durch dpkg-source --commit	28
5	Grundlagen	30
5.1	Arbeitsablauf des Paketierens	30
5.1.1	Das Paket debhelper	32
5.2	Historical perspective of Debian packaging practices	32
5.3	Future perspective on Debian packaging practices	33
5.4	Paketname und -version	34
5.5	Natives Debian-Paket	35
5.6	debian/rules	35
5.6.1	dh	36
5.6.2	Einfache debian/rules	36
5.6.3	Angepasste debian/rules	37
5.6.4	Variablen für debian/rules	37
5.6.5	Reproduzierbares Bauen	38
5.7	debian/control	38

5.7.1	Aufteilung eines Debian-Binärpakets	39
5.7.1.1	debmake -b	39
5.7.1.2	Pakettrennungsszenarien und -beispiele	40
5.7.1.3	Der Bibliothekspaketname	40
5.7.2	Substvar	41
5.7.3	biNMU-sicher	41
5.8	debian/changelog	42
5.9	debian/copyright	43
5.10	debian/patches/*	43
5.10.1	dpkg-source -x	45
5.10.2	dquilt und dpkg-source	45
5.11	debian/upstream/signing-key.asc	45
5.12	debian/watch und DFSG	46
5.13	Weitere debian/*-Dateien	46
5.14	Anpassung der Debian-Paketierung	50
5.15	Aufzeichnen im VCS (standard)	51
5.16	Aufzeichnen im VCS (Alternative)	51
5.17	Pakete ohne irrelevante Inhalte bauen	52
5.17.1	Korrektur durch debian/rules clean	52
5.17.2	Korrektur mittels VCS	53
5.17.3	Korrektur durch extend-diff-ignore	53
5.17.4	Korrektur durch tar-ignore	53
5.18	Bausysteme der Originalautoren	54
5.18.1	Autotools	54
5.18.2	CMake	55
5.18.3	Meson	55
5.18.4	Python Distutils	55
5.19	Fehlersuchinformationen	56
5.19.1	Neue -dbgsym-Pakete (Stretch 9.0 und danach)	56
5.20	Bibliothekspaket	56
5.20.1	Bibliothekssymbole	57
5.20.2	Bibliotheksübergänge	58
5.21	Debconf	58
5.22	Multiarch	59
5.22.1	The multiarch library path	59
5.22.2	The multiarch header file path	60
5.22.3	The multiarch *.pc file path	60
5.23	Compiler hardening	61
5.24	Continuous integration	61
5.25	Other new requirements	61
5.26	Bootstrapping	61
5.27	Bug reports	61
6	Debmake-Optionen	63
6.1	Abkürzungs-Optionen (-a, -i)	63
6.1.1	Python-Modul	63
6.2	Snapshot upstream tarball (-d, -t)	64
6.3	debmake -cc	64
6.4	debmake -k	64
6.5	debmake -j	65
6.6	debmake -x	66
6.7	debmake -P	66
6.8	debmake -T	66

7	Tipps	67
7.1	Debdiff	67
7.2	Dget	67
7.3	Debc	67
7.4	Piuparts	67
7.5	debsign	68
7.6	dput	68
7.7	bts	68
7.8	git-buildpackage	68
7.8.1	gbp import-dscs --debsnap	69
7.8.2	Upstream git repository	69
7.9	dgit	70
7.9.1	Snapshot upstream tarball (alternative git deborig approach)	70
7.10	chroot	70
7.11	New Debian revision	72
7.12	New upstream release	73
7.12.1	uupdate + tarball	73
7.12.2	uscan	73
7.12.3	gbp	74
7.12.4	gbp + uscan	74
7.13	3.0 source format	75
7.14	CDBS	75
7.15	Build under UTF-8	76
7.16	UTF-8 conversion	76
7.17	Upload orig.tar.gz	76
7.18	Skipped uploads	77
7.19	Advanced packaging	77
7.20	Other distros	77
7.21	Debug	78
8	Weitere Beispiele	80
8.1	Cherry-pick templates	80
8.2	Kein Makefile (Shell, CLI)	82
8.3	Makefile (Shell, CLI)	88
8.4	setup.py (Python3, CLI)	91
8.5	Makefile (Shell, GUI)	95
8.6	setup.py (Python3, GUI)	98
8.7	Makefile (Paket mit einem Programm)	101
8.8	Makefile.in + configure (Paket mit einem Programm)	103
8.9	Autotools (Paket mit einem Programm)	107
8.10	CMake (Paket mit einem Programm)	110
8.11	Autotools (Paket mit mehreren Programmen)	113
8.12	CMake (multi-binary package)	118
8.13	Internationalization	122
8.14	Details	128
A	debmake(1)-Handbuchseite	129
A.1	BEZEICHNUNG	129
A.2	ÜBERSICHT	129
A.3	BESCHREIBUNG	129
A.3.1	Optionale Argumente:	129
A.4	BEISPIELE	132
A.5	HELFERPAKETE	132
A.6	CAVEAT	133
A.7	DEBUG	133
A.8	AUTOR	133
A.9	LIZENZ	134
A.10	SIEHE AUCH	134

Abstract

Dieser Anleitung »Leitfaden für Debian-Betreuer« (2021-08-19) beschreibt gewöhnlichen Debian-Benutzern und angehenden Entwicklern den Bau des Debian-Pakets mittels des Befehls **debmake**.

Diese Anleitung konzentriert sich auf die moderne Paketierart und enthält viele einfache Beispiele.

- POSIX-Shell-Skript-Paketierung
- Python3-Skript-Paketierung
- C mit Makefile/Autotools/CMake
- mehrere Binärpakete mit Laufzeitbibliotheken usw.

Diese »Anleitung für Debian-Betreuer« kann als Nachfolger des »Debian-Leitfaden für Neue Paketbetreuer« angesehen werden.

Vorwort

Falls Sie ein etwas erfahrener Debian-Benutzer¹ sind, könnten Ihnen die folgende Situationen bekannt vorkommen:

- Sie möchten ein bestimmtes, noch nicht im Debian-Archiv verfügbares Softwarepaket installieren.
- Sie möchten ein Debian-Paket mit einer neueren Version der Originalautoren aktualisieren.
- Sie möchten Fehler in einem Debian-Paket mit Patches korrigieren.

Falls Sie ein Debian-Paket erstellen möchten, um diese Wünsche zu erfüllen und Ihre Arbeit der Debian-Gemeinschaft zur Verfügung stellen wollen, dann sind Sie die Zielgruppe dieser Anleitung als angehender Debian-Betreuer.² Willkommen in der Debian-Gemeinschaft.

Debian muss vielen sozialen und technischen Regeln und Konventionen folgen, da es eine große Organisation Freiwilliger mit einer Geschichte ist. Debian hat auch eine große Anzahl an Paketierungswerkzeugen und Archivverwaltungswerkzeugen entwickelt, um eine konsistente Menge an Binärpaketen zu bauen, die viele technische Ziele adressieren:

- Paketbau unter vielen Architekturen (Section 5.6.4)
- reproduzierbarer Bau (Section 5.6.5)
- sauberer Bau unter klar festgelegten Paketabhängigkeiten und -Patches (Section 5.7, Section 5.10, Section 7.10)
- optimale Trennung in mehrere Binärpakete (Section 5.7.1)
- reibungslose Bibliotheksübergänge (Section 5.20.2)
- interaktive Anpassung der Installation (Section 5.21)
- Multiarch-Unterstützung (Section 5.22)
- Sicherheitserweiterung mittels bestimmter Compiler-Schalter (Section 5.23)
- kontinuierliche Integration (Section 5.24)
- Systemstarteinrichtung (Section 5.26)
- ...

Dies ist für viele angehende Debian-Entwickler beim Einstieg in Debian etwas überwältigend. Diese Anleitung versucht, für ihren Start Einstiegspunkte bereitzustellen. Sie beschreibt folgendes:

- Was Sie wissen sollten, bevor Sie als angehender Betreuer bei Debian mitmachen.
- Wie es aussieht, ein einfaches Debian-Paket zu erstellen.
- Welche Art von Regeln für das Erstellen des Debian-Pakets existieren.
- Tipps zur Erstellung des Debian-Pakets.
- Beispiele für das Erstellen von Debian-Paketen für mehrere typische Szenarien.

¹Sie müssen ein bisschen über Unix-Programmierung wissen, aber Sie müssen definitiv kein Meister sein. Über den grundlegenden Umgang mit einem Debian-System können Sie in [Debian-Referenz](#) mehr erfahren. Es enthält auch einige Hinweise zum Lernen der Unix-Programmierung.

²Falls Sie nicht daran interessiert sind, das Debian-Paket anderen zugänglich zu machen, können Sie natürlich Ihre lokale Situation auch dadurch beheben, dass Sie die korrigierten Quellen der Originalautoren übersetzen und in `/usr/local/` installieren.

Der Autor bemerkte Einschränkungen beim Aktualisieren des ursprünglichen »New Maintainers' Guide« mit dem Paket **dh-make** und entschied sich, ein alternatives Werkzeug und die dazugehörige Dokumentation zu erstellen, um moderne Anforderungen zu adressieren. Das Ergebnis ist das Paket **debmake** (version: 4.4.0) und dieser aktualisierte »Guide for Debian Maintainers« in dem Paket **debmake-doc** (Version: 1.17-1).

Viele Aufgaben und Tipps sind im Befehl **debmake** integriert. Daher ist die Anleitung einfach. Diese Anleitung bietet auch viele Paketierbeispiele.

Caution



Es bedarf vieler Stunden, um Debian-Pakete zu erstellen und zu betreuen. Der Debian-Betreuer muss **sowohl technisch kompetent als auch sorgfältig** sein, um diese Herausforderung zu meistern.

Einige wichtige Themen sind im Detail erklärt. Einige mögen Ihnen unwichtig erscheinen. Seien Sie geduldig. Einige Sonderfälle werden nur durch Referenzen auf externe Dokumente berücksichtigt. Dies passiert absichtlich, um diese Anleitung einfach und wartbar zu halten.

Chapter 1

Überblick

Das Debian-Paket für *package-1.0.tar.gz*, das eine einzige, einfache C-Quelle enthält, die [GNU-Programmiers-Standard](#) and [FHS](#) folgt, kann mit dem Befehl **debmake** wie folgt erstellt werden:

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

Falls die manuellen Anpassungen der erstellten Konfigurationsdateien übersprungen werden, fehlt dem erstellten Binärpaket eine sinnvolle Paketbeschreibung. Es funktioniert aber dennoch unter dem Befehl **dpkg** gut für Ihren lokalen Einsatz.

Caution



Der Befehl **debmake** stellt nur gute Vorlagendateien bereit. Diese Vorlagendateien müssen durch manuelle Anpassungen perfektioniert werden, um den strengen Qualitätsanforderungen des Debian-Archivs zu genügen, falls das Paket für die allgemeine Verwendung gedacht ist.

Falls Sie beim Debian-Paketieren neu sind, machen Sie sich keine Kopf über die Details und erkennen Sie nur das große Ganze.

Falls Sie schon Berührungen mit der Debian-Paketierung hatten, sieht dies dem Befehl **dh_make** sehr ähnlich. Dies kommt daher, dass der Befehl **debmake** dazu gedacht ist, die historisch von dem Befehl **dh_make** angebotenen Funktionen zu ersetzen. ¹

Der Befehl **debmake** wurde mit den folgenden Funktionalitäten gestaltet:

- moderner Paketierstil
 - **debian/copyright**: [DEP-5](#) compliant
 - **debian/control**: **substvar**-Unterstützung, **multiarch**-Unterstützung, mehrere Binärpakete, ...
 - **debian/rules**: **dh**-Syntax, Compiler-Härtungsoptionen, ...
- Flexibilität
 - viele Optionen (Section [5.7.1.1](#), Chapter [6](#), Appendix [A](#))
- vernünftige Standardaktionen
 - Ausführung ohne Unterbrechung mit sauberen Ergebnissen
 - Erstellung des Multiarch-Pakets, außer die Option **-m** wird explizit angegeben

¹Der Befehl **deb-make** war vor dem Befehl **dh_make** beliebt. Das aktuelle Paket **debmake** beginnt seine Version bei **4.0**, um Überlappungen mit dem veralteten Paket **debmake**, das den Befehl **deb-make** bereitstellte, zu vermeiden.

- Erstellung des nicht-nativen Debian-Pakets im Format »**3.0 (quilt)**«, außer die Option **-n** wird explizit angegeben
- zusätzliche Nützlichkeiten
 - Überprüfung der Datei **debian/copyright** mit den aktuellen Quellen (Section 6.4)

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

Tip



Stellen Sie sicher, dass die Argumente **-b**, **-f**, **-l** und **-w** durch Maskieren vor dem Einfluss der Shell geschützt sind.

Tip



Das nicht-native Debian-Paket ist das normale Debian-Paket.

Tip



Das detaillierte Protokoll aller Paketbaubeispiele in diesem Dokument kann durch Folgen der Anweisungen in Section 8.14 erhalten werden.

Note



Die Erstellung der Datei **debian/copyright** und die Ausgabe der Optionen **-c** (Section 6.3) und **-k** (Section 6.4) beinhalten heuristische Aktionen bezüglich der Copyright- und Lizenzinformationen. Sie können fehlerhafte Ausgaben erstellen.

Chapter 2

Voraussetzungen

Hier sind die Voraussetzungen, die Sie verstehen müssen, um bei Debian mitzumachen.

2.1 Leute bei Debian

Es nehmen verschiedene Arten von Leuten mit verschiedenen Rollen rund um Debian teil:

- **Originalautor:** die Person, die das ursprüngliche Programm erstellte.
- **Originalbetreuer:** Die Person, die derzeit das Programm betreut.
- **Betreuer:** Die Person, die das Debian-Paket des Programms erstellt.
- **Sponsor:** Eine Person, die den Betreuern beim Hochladen (nach Prüfen des Inhalts) des Pakets in das offizielle Debian-Paketarchiv hilft.
- **Mentor:** Eine Person, die neuen Betreuern beim Paketieren usw. hilft.
- **Debian-Entwickler (DD):** ein Mitglied des Debian-Projekts mit vollen Hochlade-Rechten in das offizielle Debian-Paketarchiv.
- **Debian-Betreuer (DM):** Eine Person, mit begrenzten Hochladerechten in das offizielle Debian-Paketarchiv.

Bitte beachten Sie, dass Sie nicht über Nacht ein offizieller **Debian-Entwickler (DD)** werden können, da es mehr als technischer Fähigkeit bedarf. Seien Sie davon nicht entmutigt. Falls es für andere nützlich ist, können Sie dennoch Ihre Pakete hochladen, entweder als **Betreuer** durch einen **Sponsor** oder als ein **Debian-Betreuer**.

Bitte beachten Sie, dass Sie keine neuen Pakete erstellen müssen, um ein offizieller Debian-Entwickler zu werden. Auch Beiträge zu bestehenden Paketen können ein Weg sein, ein offizieller Debian-Entwickler zu werden. Es gibt viele Pakete, die auf gute Betreuer warten (siehe Section 2.8).

2.2 Wie Sie beitragen können

Bitte lesen Sie die folgenden Dokumenten, um zu verstehen, wie Sie zu Debian beitragen können:

- [Wie können Sie Debian helfen?](#) (offiziell)
- [The Debian GNU/Linux FAQ, Chapter 13 - "Contributing to the Debian Project"](#) (semi-official)
- [Debian Wiki, HelpDebian](#) (ergänzend)
- [Debian New Member site](#) (offiziell)
- [Debian Mentors FAQ](#) (ergänzend)

2.3 Soziale Dynamik von Debian

Bitte verstehen Sie die soziale Dynamik, um sich für die Wechselwirkungen mit Debian vorzubereiten:

- Wir sind alle Freiwillige.
 - Sie können anderen nicht vorschreiben, was getan werden soll.
 - Sie sollten motiviert sein, die Dinge selbst zu tun.
- Freundliche Zusammenarbeit ist der Motor.
 - Ihr Beitrag sollte Andere nicht überlasten.
 - Ihr Beitrag hat nur Wert, wenn ihn andere wertschätzen.
- Debian ist nicht Ihre Schule, bei der die Lehrer sich automatisch um Sie kümmern.
 - Sie sollten vieles selbst lernen können.
 - Aufmerksamkeit von anderen Freiwilligen ist ein rares Gut.
- Debian verbessert sich ständig.
 - Es wird erwartet, dass Sie hochqualitative Pakete erstellen.
 - Sie sollten sich Änderungen anpassen.

Da in der restlichen Anleitung nur auf die technischen Aspekte der Paketierung fokussiert wird, wird auf folgendes zum Verständnis der sozialen Dynamik von Debian verwiesen:

- [Debian: 17 years of Free Software, “do-ocracy”, and democracy](#) (Einleitende Folien vom ehemaligen DPL)

2.4 Technische Erinnerungen

Es folgen ein paar technische Erinnerungen, die anderen Betreuern bei der Arbeit an Ihrem Paket helfen und das Ergebnis von Debian insgesamt effektiv maximieren.

- Ermöglichen Sie in Ihrem Paket eine leichte Fehlersuche.
 - Halten Sie Ihr Paket einfach.
 - Machen Sie Ihr Paket nicht zu komplex.
- Halten Sie Ihr Paket gut dokumentiert.
 - Verwenden Sie einen lesbaren Code-Stil.
 - Fügen Sie in den Code Kommentare ein.
 - Formatieren Sie Code konsistent.
 - Verwalten Sie das Git-Depot ¹ des Pakets.

Note



Die Fehlersuche in Software kann mehr Zeit verbrauchen, als die eigentliche Erstellung.

¹Die überwältigende Mehrheit der Debian-Betreuer verwenden **git** statt anderer VCS-Systeme wie **hg**, **bzr**, usw.

2.5 Debian-Dokumentation

Bitte lesen Sie nach Bedarf den einschlägigen Anteil der offiziellen Debian-Dokumentation zusammen mit dieser Anleitung, um perfekte Debian-Pakete zu erstellen:

- Debian-Policy Handbuch (englisch)
 - Regeln, den Sie folgen müssen (<https://www.debian.org/doc/devel-manuals#policy>)
- Debian-Entwickler-Referenz
 - Dokument zur »Besten Vorgehensweise« (<https://www.debian.org/doc/devel-manuals#devref>)

Falls dieser Leitfaden der offiziellen Debian-Dokumentation widerspricht, dann ist die offizielle Debian-Dokumentation korrekt. Bitte reichen Sie mittels des Befehls **reportbug** (auf englisch) einen Fehlerbericht gegen das Paket **debmake-doc** ein.

Hier sind alternative Anleitungen, die Sie zusammen mit diesem Leitfaden lesen können:

- »Debian-Leitfaden für Neue Paketbetreuer« (älter)
 - <https://www.debian.org/doc/devel-manuals#maint-guide>
 - <https://packages.qa.debian.org/m/maint-guide.html>
- Leitfaden zum Debian-Paketieren
 - <https://www.debian.org/doc/devel-manuals#packaging-tutorial>
 - <https://packages.qa.debian.org/p/packaging-tutorial.html>
- »Ubuntu Packaging Guide« (Ubuntu basiert auf Debian.) (englisch)
 - <http://packaging.ubuntu.com/html/>

Tip



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

2.6 Hilfequellen

Bevor Sie Ihre Fragen an einem öffentlichen Ort stellen, bemühen Sie sich vorher, die Antwort zu ermitteln, z.B. lesen Sie die gute Dokumentation:

- Paketierinformationen, die über die Befehle **aptitude**, **apt-cache** und **dpkg** verfügbar sind.
- Dateien in **/usr/share/doc/Paket** für alle einschlägigen Pakete.
- Inhalte von **man Befehl** für alle einschlägigen Befehle.
- Inhalte von **info Befehl** für alle einschlägigen Befehle.
- Inhalte von debian-mentors@lists.debian.org-Mailinglist-Archiv.
- Inhalte von debian-devel@lists.debian.org-Mailinglist-Archiv.

Ihre gewünschten Informationen können effektiv gefunden werden, indem Sie einen wohlgeformten Suchbegriff wie das Schlüsselwort **site:lists.debian.org** verwenden, um den Suchbereich der Web-Suchmaschine einzuschränken.

Die Herstellung eines kleinen Testpaketes ist eine gute Möglichkeit, Details des Paketierens zu erfahren. Bestehende, gut gewartete Debian-Pakete zu inspizieren ist der beste Weg, um zu lernen, wie andere Menschen Debian-Pakete erstellen.

Falls Sie immer noch Fragen über das Paketieren haben, können Sie die Fragen direkt stellen (auf Englisch):

- debian-mentors@lists.debian.org Mailingliste. (Diese Mailingliste ist für die Anfänger.)
- debian-devel@lists.debian.org Mailingliste. (Diese Mailingliste ist für die Experten.)
- IRC wie #debian-mentors.
- Teams, die sich auf eine bestimmte Reihe von Paketen konzentrieren. (Vollständige Liste unter <https://wiki.debian.org/Teams>)
- Sprachspezifische Mailinglisten
 - debian-devel-{french,italian,portuguese,spanish}@lists.debian.org
 - debian-chinese-gb@lists.debian.org (Diese Mailingliste ist für allgemeine Diskussion auf (vereinfachtem) Chinesisch.)
 - debian-devel@debian.or.jp

Die erfahreneren Debian-Entwickler werden Ihnen gerne helfen, wenn Sie nach den erforderlichen Vorarbeiten richtig fragen.

Caution



Die Debian-Entwicklung ist ein sich entwickelndes Projekt. Einige Informationen, die im Web gefunden werden, könnten veraltet, falsch und nicht (mehr) zutreffend sein. Bitte benutzen Sie sie sorgfältig.

2.7 Archivsituation

Bitte begreifen Sie die Situation des Debian-Archivs.

- Debian hat bereits Pakete für die meisten Programmarten.
- Die Anzahl an Paketen im Debian-Archiv ist mehrere zehnmal so groß wie die Anzahl der aktiven Betreuer.
- Bei einigen Paketen fehlt leider die entsprechende Aufmerksamkeit des Betreuers.

Daher werden Beiträge zu Paketen, die sich bereits im Archiv befinden, von anderen Betreuern weitaus mehr geschätzt (und erhalten eher Sponsoring für das Hochladen).

Tip



Der Befehl **wnpp-alert** aus dem Paket **devscripts** kann prüfen, ob installierte Pakete zur Adoption freigegeben oder verwaist sind.

2.8 Wege, beizutragen

Hier ist Pseudo-Python-Code für Ihre Wege, zu Debian mit einem **Programm** beizutragen:

```
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)

```

Hier:

- Für das Prüfen von `exist_in_debian()` und `is_team_maintained()`:
 - der Befehl **aptitude**
 - [Debian-Pakete](#)-Webseite
 - [Teams](#)
- Für das Prüfen von `is_orphaned()`, `is_RFA()` und `is_ITPed_by_others()`:
 - Die Ausgabe des Befehls **wnpp-alert**.
 - [Arbeit-bedürfende und voraussichtliche Pakete](#)
 - [Debian-Fehlerberichtsprotokolle: Fehler im Pseudo-Paket wnpp in Unstable](#)
 - [Debian-Pakete, die lieb gewonnen werden müssen](#)
 - [Durchsuchen Sie wnpp Fehlerberichte basierend auf Debtags](#)
- Für das Prüfen von `is_good_program()`:
 - Das Programm sollte nützlich sein.
 - Das Programm sollte keine Sicherheits- und Wartungsbedenken in das Debian-System einbringen.
 - Das Programm sollte gut dokumentiert sein und sein Code muss verständlich (d.h. nicht verschleiert) sein.
 - Die Autoren des Programms stimmen mit der Paketierung überein und sind mit Debian einvernehmlich.²
- Für die Prüfung von `is_it_DFSG()` und `is_its_dependency_DFSG()`:
 - [Debian Richtlinien für Freie Software](#) (DFSG).
- Für die Prüfung von `is_it_distributable()`:
 - Die Software muss eine Lizenz haben und sie sollte ihre Verbreitung erlauben.

²Das ist nicht die absolute Voraussetzung. Die ablehnende Einstellung der Originalautoren kann für uns alle zu einem großen Ressourcenproblem werden. Die freundlichen Originalautoren können konsultiert werden, um alle Probleme mit dem Programm zu lösen.

Sie müssen entweder einen **ITP** einreichen oder ein Paket adoptieren, um mit der Arbeit daran zu beginnen. Siehe die »Debian-Entwickler-Referenz«:

- [5.1. Neue Pakete.](#)
- [5.9. Verschieben, Entfernen, Verwaisen, Adoptieren und Wiedereinführen von Paketen.](#)

2.9 Neue Beitragende und Betreuer

Die neuen Beitragenden und Betreuer könnten sich fragen, was sie lernen sollten, um zu Debian beizutragen. Hier sind meine Vorschläge, abhängig von Ihrem Schwerpunkt:

- Paketierung
 - Grundlagen der **POSIX-Shell** und **Make**.
 - Etwas rudimentäres Wissen von **Perl** und **Python**.
- Übersetzungen
 - Grundlagen, wie das PO-basierte Übersetzungssystem funktioniert.
- Dokumentation
 - Grundlagen der Textauszeichnung (XML, ReST, Wiki, ...).

Der neue Beitragende und Betreuer könnte sich fragen, wo er mit Beiträgen zu Debian beginnen sollte. Hier sind meine Vorschläge, abhängig von Ihren Fertigkeiten:

- **POSIX-Shell**-, **Perl**- und **Python**-Fertigkeiten:
 - Schicken Sie Patches zum Debian-Installer.
 - Send patches to the Debian packaging helper scripts such as **devscripts**, **sbuild**, **schroot**, etc. mentioned in this document.
- **C**- und **C++**-Fähigkeiten:
 - Schicken Sie Patches an Pakete mit den Prioritäten **required** und **important**.
- Sprachfähigkeiten jenseits von Englisch:
 - Schicken Sie Patches zu den PO-Dateien des Debian-Installers.
 - Schicken Sie Patches an den PO-Dateien der Pakete mit den Prioritäten **required** und **important**.
- Dokumentations-Fähigkeiten
 - Aktualisieren Sie Inhalte auf [Debian-Wiki](#).
 - Schicken Sie Patches an das existierende [Debian-Dokumentation](#).

Diese Aktivitäten sollten Ihnen einen guten Kontakt zu den anderen Debian-Menschen verschaffen, um Ihre Glaubwürdigkeit zu begründen.

Der unerfahrene Betreuer sollte das Paketieren von Programmen mit einer hohen Sicherheitsexposition vermeiden:

- **Setuid**- oder **Setgid**-Programme
- **Daemon**-Programme
- Programme, die in den Verzeichnissen **/sbin/** oder **/usr/sbin/** installiert werden

Wenn Sie mehr Erfahrung beim Paketieren gesammelt haben, werden Sie in der Lage sein, solche Programme zu paketieren.

Chapter 3

Werkzeugeinrichtung

Das Paket **build-essential** muss in Ihrer Bauumgebung installiert sein.

Das Paket **devscripts** sollte in der Betreuerumgebung installiert werden.

Although this is not necessarily an absolute requirement, it is a good idea to install and set up all of the popular set of packages mentioned in this chapter in the maintainer environment. This enables us to share the common baseline working environment. . Please install the tools mentioned in the [Overview of Debian Maintainer Tools](#) in the “Debian Developer’s Reference”, as needed, too.

Caution



Die hier vorgestellte Einrichtung der Werkzeuge ist nur als Beispiel gedacht und kann nicht mit den neuesten im System installierten Paketen übereinstimmen. Debian-Entwicklung ist ein voranschreitendes Ziel. Lesen Sie die relevante Dokumentation und aktualisieren Sie Ihre Konfiguration wo notwendig.

3.1 E-Mail-Adresse

Verschiedene Debian-Betreuungswerkzeuge erkennen Ihre zu verwendene E-Mail-Adresse und Ihren Namen über die Shell-Umgebungsvariablen **\$DEBEMAIL** und **\$DEBFULLNAME**.

Richten Sie diese Pakete ein, indem Sie die nachfolgenden Zeilen zu **~/.bashrc** hinzufügen ¹.

Fügen Sie zur **~/.bashrc folgendes hinzu**

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

3.2 MC

Der Befehl **mc** bietet Ihnen einen sehr leichten Zugang zur Verwaltung von Dateien. Es kann binäre **deb**-Dateien zum Prüfen ihrer Inhalte öffnen, indem die Eingabetaste über der binären **deb**-Datei gedrückt wird. Es verwendet den Befehl **dpkg-deb** als seinen Backend. Lassen Sie es uns wie folgt einrichten, um leichtes **chdir** zu ermöglichen.

Fügen Sie zur **~/.bashrc folgendes hinzu**

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

¹Hierbei wird angenommen, dass Sie die Bash als Ihre Anmelde-Shell verwenden. Falls Sie eine andere Anmelde-Shell, wie die Z-Shell, verwenden, verwenden Sie deren entsprechenden Konfigurationsdateien statt **~/.bashrc**.

3.3 Git

Heutzutage ist der Befehl **git** ein unverzichtbares Werkzeug, um den Quellbau mit Verlauf zu verwalten.

Die globale Benutzerkonfiguration für den Befehl **git**, wie Ihren Namen und Ihre E-Mail-Adresse, können in `~/.gitconfig` wie folgt gesetzt werden:

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

Falls Sie an die CVS- oder Subversion-Befehle gewöhnt sind, möchten Sie vielleicht eine Reihe von Aliassen wie folgt setzen:

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

Sie können Ihre globale Konfiguration wie folgt prüfen:

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

Tip



Es ist essentiell, einige GUI git-Werkzeuge wie **gitk** oder **gitg** zu verwenden, um effektiv mit der Chronik des git-Depots zu arbeiten.

3.4 Quilt

Der Befehl **quilt** bietet eine grundlegende Methode zur Aufzeichnung von Änderungen. Für die Debian-Paketierung sollte es so angepasst werden, dass Änderungen im **debian/patches/**-Verzeichnis anstelle des voreingestellten **patches/**-Verzeichnisses aufgezeichnet werden.

Um zu vermeiden, dass das Verhalten des Befehls **quilt** selbst geändert wird, erstellen wir einen Alias **dquilt** für das Debian-Paket, indem wir die folgenden Zeilen in die Datei `~/.bashrc` einfügen. Die zweite Zeile bietet die gleiche Shell-Vervollständigungs-Funktion des **quilt**-Befehls für den **dquilt**-Befehl.

Fügen Sie zur `~/.bashrc` folgendes hinzu

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

Dann erstellen wir `~/.quiltrc-dpkg` wie folgt.

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

Siehe **quilt(1)** und [Wie man mit vielen Patches überlebt oder eine Einführung in Quilt](#), wie man den Befehl **quilt** verwendet.

Siehe Section 4.9 für beispielhafte Verwendungen.

3.5 Devscripts

Der Befehl **debsign**, der im Paket **devscripts** enthalten ist, wird verwendet, um das Debian-Paket mit Ihrem privaten GPG-Schlüssel zu signieren.

Der Befehl **debuild**, der im Paket **devscripts** enthalten ist, baut das Binärpaket und überprüft es mit dem Befehl **lintian**. Es ist praktisch, ausführliche Ausgaben von dem Befehl **lintian** zu haben.

Sie können diese in `~/.devscripts` wie folgt einrichten.

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

Die Optionen `-` und `-I` in **DEBUILD_DPKG_BUILDPACKAGE_OPTS** für den Befehl **dpkg-source** helfen beim erneuten Bauen von Debian-Paketen ohne Fremdinhalte (siehe Section 5.17).

Derzeit ist ein RSA-Schlüssel mit 4096 Bits eine gute Idee. Siehe [Einen neuen GPG-Key erstellen](#).

3.6 sbuild

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 [build](#) infrastructure, it is always up to date and comes with full of useful features.

It can be customized to offer following features:

- The **schroot** package to boost the chroot creation speed.
- Das **lintian**-Paket, um Fehler im Paket zu finden.
- The **piuparts** package to find bugs in the package.
- The **autopkgtest** package to find bugs in the package.
- Das **ccache**-Paket, um die **gcc**-Geschwindigkeit zu erhöhen. (optional)
- Das **libeatmydata1**-Paket zur Erhöhung der **dpkg**-Geschwindigkeit. (optional)
- Die parallele **make**, um die Baugeschwindigkeit zu erhöhen. (optional)

Let's get started by following <https://wiki.debian.org/sbuild>.

```
$ sudo apt install sbuild piuparts autopkgtest lintian
$ adduser <your_user_name> sbuild
```

Logout and login to check you are a member of sbuild group using `id` command.

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

Let's create the configuration file `~/.sbuilderdrc` in line with recent Debian practice <https://wiki.debian.org/SourceOnlyUpload> as:

```
cat >~/.sbuilderdrc << 'EOF'
#####
# PACKAGE BUILD RELATED (source-only-upload as default)
#####
# -d
$distribution = 'unstable';
# -A
$sbuild_arch_all = 1;
# -s
$sbuild_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
```

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,

Warning



Die optionale Anpassung kann negative Auswirkungen haben. Im Zweifelsfall deaktivieren Sie sie.

Note



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

3.7 git-buildpackage

Es kann sinnvoll sein, dass Sie mehrere globale Konfigurationen in `~/gbp.conf` setzen.

```
# 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
```

Tip



Der Befehl **gbp** ist ein Alias für den Befehl **git-buildpackage**.

3.8 HTTP-Proxy

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

- 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.

3.9 Privates Debian-Depot

Sie können ein privates Debian-Depot mit dem Paket **reprepro** einrichten.

3.10 Virtual machines

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

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

3.11 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 4

Einfaches Beispiel

Es gibt ein altes lateinisches Sprichwort: »**Longum iter est per praecepta, breve et efficax per exempla**« (»Der Weg mit den Regeln ist lang, mit Beispielen aber kurz und effizient«).

4.1 Packaging tarball

Hier ist ein Beispiel zur Erstellung eines einfachen Debian-Pakets aus einer einfachen C-Quelle mittels einer **Makefile** als Bausystem.

Nehmen wir an, der Tarball der Originalautoren heißt **debhello-0.0.tar.gz**.

Diese Art der Quellen soll als Nichtsystemdatei wie folgt installiert werden:

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

Die Debian-Paketierung ändert diesen »**make install**«-Prozess in die Installation von Dateien an den Ort im Zielsystemabbild statt des normalen Ortes unterhalb von **/usr/local**.

Note



Beispiele zur Erstellung von Debian-Paketen aus anderen, komplizierteren Bausystemen werden in [Chapter 8](#) beschrieben.

4.2 Gesamtbild

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

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

Big picture of package building

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

Tip



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

Tip



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

4.3 Was ist Debmake?

Der Befehl **debmake** ist ein Hilfsskript für die Debian-Paketierung.

- It always sets most of the obvious option states and values to reasonable defaults.
- Es erstellt den Tarball der Originalautoren und die benötigten Symlinks, falls sie fehlen.
- Es setzt die bestehenden Konfigurationsdateien im Verzeichnis **debian/** nicht außer Kraft.
- Es unterstützt die **Multiarch**-Pakete.
- Es erstellt gute Vorlagendateien wie eine **debian/copyright**-Datei, die konform zu **DEP-5** ist.

Diese Funktionalitäten ermöglichen ein einfaches und moderens Paketieren für Debian mit **debmake**.

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

Note



Many packages are packaged using only a text editor while imitating how other similar packages are packaged and consulting how the Debian policy requires us to do. This seems to me the most popular method for the real-life packaging activity.

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

4.4 Was ist Debuild?

Hier ist eine Zusammenfassung von Befehlen, die dem Befehl **debuild** ähnlich sind.

- Die Datei **debian/rules** definiert, wie das binäre Debian-Paket gebaut wird.
- Der Befehl **dpkg-buildpackage** ist der offizielle Befehl, ein Debian-Binärpaket zu bauen. Bei einem normalen Binärbau führt er grob die folgenden Schritte aus:
 - »**dpkg-source --before-build**« (wendet Debian Patches an, außer sie sind bereits angewandt)
 - »**fakeroot debian/rules clean**«
 - »**dpkg-source --build**« (baut das Debian-Quellpaket)
 - »**fakeroot debian/rules build**«
 - »**fakeroot debian/rules binary**«
 - »**dpkg-genbuildinfo**« (erstellt eine ***.buildinfo**-Datei)
 - »**dpkg-genchanges**« (erstellt eine ***.changes**-Datei)
 - »**fakeroot debian/rules clean**«
 - »**dpkg-source --after-build**« (unapply Debian patches, if they are applied during **--before-build**)
 - »**debsign**« (sign the ***.dsc** and ***.changes** files)
 - * If you followed Section 3.5 to set the **-us** and **-uc** options, this step is skipped and you must run the **debsign** command manually.
- The **debuild** command is a wrapper script of the **dpkg-buildpackage** command to build the Debian binary package under the proper environment variables.
- The **sbuild** command is a wrapper script to build the Debian binary package under the proper chroot environment with the proper environment variables.

Note



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

4.5 Schritt 1: Holen der Quellen der Originalautoren

Let's get the upstream source.

Download debhello-0.0.tar.gz

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

2 directories, 4 files
```

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

hello.c

```
$ cat debhello-0.0/src/hello.c
#include <stdio.h>
int
main()
{
    printf("Hello, world!\n");
    return 0;
}
```

Here, the **Makefile** supports [GNU Coding Standards](#) and [FHS](#). Notably:

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

Makefile

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

all: src/hello

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

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

clean:
    -rm -f src/hello

distclean: clean

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

.PHONY: all install clean distclean uninstall
```

Note



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

4.6 Schritt 2: Mit Debmake erstellte Vorlagendateien

Tip



Falls der Befehl **debmake** mit der Option **-T** aufgerufen wird, werden in den Vorlagendateien ausführlichere Kommentare erstellt.

Die Ausgabe des Befehls **debmake** ist sehr ausführlich und erklärt seine Tätigkeiten wie folgt:

```
$ cd debhello-0.0
$ debmake
I: set parameters
I: =====
I: package_dir      = /usr/lib/python3/dist-packages
I: base_path        = /usr
I: base_lib_path     = /usr/lib/debmake
I: base_share_path   = /usr/share/debmake
I: =====
I: sanity check of parameters
I: pkg="debhello", ver="0.0", rev="1"
I: *** start packaging in "debhello-0.0". ***
I: provide debhello_0.0.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-0.0.tar.gz debhello_0.0.orig.tar.gz
I: pwd = "/path/to/debhello-0.0"
I: parse binary package settings:
I: binary package=debhello Type=bin / Arch=any M-A=foreign
I: analyze the source tree
I: build_type = make
I: scan source for copyright+license text and file extensions
I: 100 %, ext = c
I: check_all_licenses
I: ..
I: check_all_licenses completed for 2 files.
I: bunch_all_licenses
I: format_all_licenses
I: make debian/* template files
I: single binary package
I: debmake -x "1" ...
I: creating => debian/control
I: creating => debian/copyright
I: substituting => /usr/share/debmake/extra0/changelog
I: creating => debian/changelog
I: substituting => /usr/share/debmake/extra0/rules
I: creating => debian/rules
I: substituting => /usr/share/debmake/extra1/README.Debian
I: creating => debian/README.Debian
I: substituting => /usr/share/debmake/extra1/watch
I: creating => debian/watch
I: substituting => /usr/share/debmake/extra1source/format
I: creating => debian/source/format
I: substituting => /usr/share/debmake/extra1tests/control
I: creating => debian/source/control
I: substituting => /usr/share/debmake/extra1upstream/metadata
I: creating => debian/upstream/metadata
I: substituting => /usr/share/debmake/extra1tests/control
I: creating => debian/tests/control
I: substituting => /usr/share/debmake/extra1patches/series
I: creating => debian/patches/series
I: substituting => /usr/share/debmake/extra1sourcex/local-options
I: creating => debian/source/local-options
I: substituting => /usr/share/debmake/extra1sourcex/options
I: creating => debian/source/options
I: substituting => /usr/share/debmake/extra1sourcex/patch-header
I: creating => debian/source/patch-header
I: run "debmake -x2" to get more template files
I: $ wrap-and-sort
```

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

- Der Quellpaketname: **debhello**

- Die Version der Originalautoren: **0.0**
- Deb Binärpaketname: **debhello**
- The Debian revision: **1**
- The package type: **bin** (the ELF binary executable package)
- Die Option **-x: -x1** (Vorgabe für das einzelne Binärpaket)

Lassen Sie uns die erstellten Vorlagendateien anschauen.

Der Quellbaum, nach der grundlegenden Ausführung von debmake.

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

7 directories, 19 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 (Vorlagendatei):

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

%:
    dh $@

#override_dh_auto_install:
#    dh_auto_install -- prefix=/usr

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

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

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

debian/control (Vorlagendatei):

```
$ cat debhello-0.0/debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: "Firstname Lastname" <email.address@example.org>
Build-Depends: debhelper-compat (= 13)
Standards-Version: 4.5.1
Homepage: <insert the upstream URL, if relevant>
Rules-Requires-Root: no

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

Warning



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

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

Note



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

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

debian/copyright (Vorlagendatei):

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

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

#-----
# Files marked as NO_LICENSE_TEXT_FOUND may be covered by the following
```

```
# license/copyright files.

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

4.7 Schritt 3: Anpassung der Vorlagendateien

Um ein geeignetes Debian-Paket zu erstellen, sind manuelle Anpassungen durch den Betreuer notwendig.

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 (Betreuerversion):

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

%:
    dh $@

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

Exportieren der Umgebungsvariable **DH_VERBOSE** in der Datei **debian/rules** zwingt das Werkzeug **debhelper** dazu, einen feingranularen Baubericht zu erstellen.

Durch Exportieren von **DEB_BUILD_MAINT_OPTION** wie oben werden die Härtungsoptionen wie in »FUNKTIONALITÄTSBEREICHE« in **dpkg-buildflags(1)** beschrieben gesetzt.¹

Durch Exportieren von **DEB_CFLAGS_MAINT_APPEND** wie oben wird der C-Compiler gezwungen, alle Warnungen auszugeben.

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.

Durch Exportieren von **DEB_LDFLAGS_MAINT_APPEND** wie oben wird der Linker gezwungen, nur zu linken, wenn die Bibliothek tatsächlich benötigt wird.²

Der Befehl **dh_auto_install** für das Makefile-basierende Bausystem führt im wesentlichen »**\$(MAKE) install DESTDIR=debian/debhello**« aus. Die Erstellung dieses Zieles **override_dh_auto_install** ändert sein Verhalten zu »**\$(MAKE) install DESTDIR=debian/debhello prefix=/usr**«.

Hier sind die Betreuerversionen der Dateien **debian/control** und **debian/copyright**.

debian/control (Betreuerversion):

```
$ vim debhello-0.0/debian/control
... hack, hack, hack, ...
$ cat debhello-0.0/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper-compat (= 13)
Standards-Version: 4.5.1
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)
```

debian/copyright (Betreuerversion):

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

Es gibt eine Reihe von weiteren Vorlagendateien unter dem Verzeichnis **debian/**. Diese müssen auch aktualisiert werden.

Vorlagendateien unter debian/. (v=0.0):

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.

```
$ tree debhello-0.0/debian
debhello-0.0/debian
b'|-b'|-b'|-b'|-b' README.Debian
b'|-b'|-b'|-b'|-b' changelog
b'|-b'|-b'|-b'|-b' control
b'|-b'|-b'|-b'|-b' copyright
b'|-b'|-b'|-b'|-b' patches
b'|-b'|-b'|-b'|-b' series
b'|-b'|-b'|-b'|-b' rules
b'|-b'|-b'|-b'|-b' source
b'|-b'|-b'|-b'|-b' control
b'|-b'|-b'|-b'|-b' format
b'|-b'|-b'|-b'|-b' local-options
b'|-b'|-b'|-b'|-b' options
b'|-b'|-b'|-b'|-b' patch-header
b'|-b'|-b'|-b'|-b' tests
b'|-b'|-b'|-b'|-b' control
b'|-b'|-b'|-b'|-b' upstream
b'|-b'|-b'|-b'|-b' metadata
b'|-b'|-b'|-b'|-b' watch

4 directories, 14 files
```

Tip



Konfigurationsdateien, die vom Befehl **dh_*** aus dem Paket **debhelper** verwandt werden, behandeln normalerweise **#** als Beginn einer Kommentarzeile.

4.8 Schritt 4: Paketbau mit debuild

You can create a non-native Debian package using the **debuild** command or its equivalents (see Section 4.4) in this source tree. The command output is very verbose and explains what it does as follows.

```
$ cd debhello-0.0
$ debuild
dpkg-buildpackage -us -uc -ui -i -i
...
debian/rules clean
dh clean
...
debian/rules binary
dh binary
  dh_update_autotools_config
  dh_autoreconf
  dh_auto_configure
    install -d /path/to/debhello-0.0/debian/.debhelper/generated/_source/...
  dh_auto_build
    make -j12 "INSTALL=install --strip-program=true"
make[1]: Entering directory '/path/to/debhello-0.0'
# CFLAGS=-g -O2
# -ffile-prefix-map=/home/osamu/src/public/debmake-doc/debmake-doc/examp
...
Now running lintian -i -I --show-overrides debhello_0.0-1_amd64.changes ...
...
N:   Renamed from: binary-without-manpage
N:
W: debhello: readme-debian-contains-debmake-template
N:
```


...

Sie können überprüfen, dass **CFLAGS** korrekt mit **-Wall** und **-pedantic** durch die Variable **DEB_CFLAGS_MAINT_APPEND** aktualisiert ist.

Wie vom Paket **lintian** berichtet sollte eine Handbuchseite zum Paket hinzugefügt werden, wie dies in späteren Beispielen gezeigt wird (siehe Chapter 8). Lassen Sie uns jetzt weitermachen.

Lassen Sie uns die Ergebnisse anschauen.

Die durch den Befehl `debuild` erstellten Dateien von `debhello` Version 0.0:

```
$ cd ..
$ tree -FL 1
.
|_b''b''-b''b''-b'' debhello-0.0/
|_b''b''-b''b''-b'' debhello-0.0.tar.gz
|_b''b''-b''b''-b'' debhello-dbgsym_0.0-1_amd64.deb
|_b''b''-b''b''-b'' debhello_0.0-1.debian.tar.xz
|_b''b''-b''b''-b'' debhello_0.0-1.dsc
|_b''b''-b''b''-b'' debhello_0.0-1_amd64.build
|_b''b''-b''b''-b'' debhello_0.0-1_amd64.buildinfo
|_b''b''-b''b''-b'' debhello_0.0-1_amd64.changes
|_b''b''-b''b''-b'' debhello_0.0-1_amd64.deb
|_b''b''-b''b''-b'' debhello_0.0.orig.tar.gz -> debhello-0.0.tar.gz

1 directory, 9 files
```

You see all the generated files.

- Das **debhello_0.0.orig.tar.gz** ist ein Symmlink auf den Tarball der Originalautoren.
- The **debhello_0.0-1.debian.tar.xz** contains the maintainer generated contents.
- The **debhello_0.0-1.dsc** is the meta data file for the Debian source package.
- The **debhello_0.0-1_amd64.deb** is the Debian binary package.
- The **debhello-dbgsym_0.0-1_amd64.deb** is the Debian debug symbol binary package. See Section 5.19.1.
- The **debhello_0.0-1_amd64.build** file is the build log file.
- The **debhello_0.0-1_amd64.buildinfo** file is the meta data file generated by **dpkg-genbuildinfo(1)**.
- The **debhello_0.0-1_amd64.changes** is the meta data file for the Debian binary package.

The **debhello_0.0-1.debian.tar.xz** contains the Debian changes to the upstream source as follows.

The compressed archive contents of `debhello_0.0-1.debian.tar.xz`:

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

```

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-debsym_0.0-1_amd64.deb** contains the debug symbol files to be installed to the target system..

The binary package contents of all binary packages:

```

$ dpkg -c debhello-dbgSYM_0.0-1_amd64.deb
drwxr-xr-x root/root ... ./
drwxr-xr-x root/root ... ./usr/
drwxr-xr-x root/root ... ./usr/lib/
drwxr-xr-x root/root ... ./usr/lib/debug/
drwxr-xr-x root/root ... ./usr/lib/debug/.build-id/
drwxr-xr-x root/root ... ./usr/lib/debug/.build-id/be/
-rw-r--r-- root/root ... ./usr/lib/debug/.build-id/be/11292eded3fc22396a0b62...
drwxr-xr-x root/root ... ./usr/share/
drwxr-xr-x root/root ... ./usr/share/doc/
lrwxrwxrwx root/root ... ./usr/share/doc/debhello-dbgSYM -> debhello
$ dpkg -c debhello_0.0-1_amd64.deb
drwxr-xr-x root/root ... ./
drwxr-xr-x root/root ... ./usr/
drwxr-xr-x root/root ... ./usr/bin/
-rwxr-xr-x root/root ... ./usr/bin/hello
drwxr-xr-x root/root ... ./usr/share/
drwxr-xr-x root/root ... ./usr/share/doc/
drwxr-xr-x root/root ... ./usr/share/doc/debhello/
-rw-r--r-- root/root ... ./usr/share/doc/debhello/README.Debian
-rw-r--r-- root/root ... ./usr/share/doc/debhello/changelog.Debian.gz
-rw-r--r-- root/root ... ./usr/share/doc/debhello/copyright

```

The generated dependency list of all binary packages.

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

```

$ dpkg -f debhello-dbgSYM_0.0-1_amd64.deb pre-depends \
    depends recommends conflicts breaks
Depends: debhello (= 0.0-1)
$ dpkg -f debhello_0.0-1_amd64.deb pre-depends \
    depends recommends conflicts breaks
Depends: libc6 (>= 2.2.5)

```

Caution



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

Note



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

4.9 Schritt 3 (alternativ): Änderungen der Quellen der Originalautoren

Das obige Beispiel veränderte die Quellen der Originalautoren nicht, um ein geeignetes Debian-Paket zu erstellen.

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

The packaging is practically the same as the above step-by-step example except for two points in Section 4.7:

- Store the maintainer modifications to the upstream source as the corresponding patch files in the **debian/patches/** directory and list their filenames in the **debian/patches/series** file as indicated in Section 5.10. There are several ways to generate patch files. A few examples are given in these sections:

- Section 4.9.1
- Section 4.9.2
- Section 4.9.3

- The maintainer modification to the **debian/rules** file doesn't have the **override_dh_auto_install** target as follows:

debian/rules (alternative maintainer version):

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

%:
    dh $@
```

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

Note



For this particular packaging case, the above Section 4.7 using the **debian/rules** file is the better approach. But let's keep on with this approach as a leaning process.

Tip



For more complicated packaging cases, both Section 4.7 and Section 4.9 approaches need to be deployed.

4.9.1 Patch durch diff -u

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

```
$ cp -a debhello-0.0 debhello-0.0.orig
$ vim debhello-0.0/Makefile
... hack, hack, hack, ...
$ diff -Nru debhello-0.0.orig debhello-0.0 >000-prefix-usr.patch
$ cat 000-prefix-usr.patch
```

```
diff -Nru debhello-0.0.orig/Makefile debhello-0.0/Makefile
--- debhello-0.0.orig/Makefile    2021-07-02 16:26:38.734722687 +0900
+++ debhello-0.0/Makefile         2021-07-02 16:26:38.802723496 +0900
@@ -1,4 +1,4 @@
-prefix = /usr/local
+prefix = /usr

all: src/hello

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

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

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

```
$ cd debhELLO-0.0
$ echo '000-prefix-usr.patch' >debian/patches/series
$ vim ../000-prefix-usr.patch
... hack, hack, hack, ...
$ mv -f ../000-prefix-usr.patch debian/patches/000-prefix-usr.patch
$ cat debian/patches/000-prefix-usr.patch

From: Osamu Aoki <osamu@debian.org>
Description: set prefix=/usr patch
diff -Nru debhELLO-0.0.orig/Makefile debhELLO-0.0/Makefile
--- debhELLO-0.0.orig/Makefile
+++ debhELLO-0.0/Makefile
@@ -1,4 +1,4 @@
-prefix = /usr/local
+prefix = /usr

all: src/hello
```

4.9.2 Patch durch Dquilt

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

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

```

b''|b''b''-b''b''-b'' debian
b''|b'' b''|b''b''-b''b''-b'' README.Debian
b''|b'' b''|b''b''-b''b''-b'' changelog
b''|b'' b''|b''b''-b''b''-b'' control
b''|b'' b''|b''b''-b''b''-b'' copyright
b''|b'' b''|b''b''-b''b''-b'' patches
b''|b'' b''|b'' b''|b''b''-b''b''-b'' 000-prefix-usr.patch
b''|b'' b''|b'' b''|b''b''-b''b''-b'' series
b''|b'' b''|b''b''-b''b''-b'' rules
b''|b'' b''|b''b''-b''b''-b'' source
b''|b'' b''|b'' b''|b''b''-b''b''-b'' control
b''|b'' b''|b'' b''|b''b''-b''b''-b'' format
b''|b'' b''|b'' b''|b''b''-b''b''-b'' local-options
b''|b'' b''|b'' b''|b''b''-b''b''-b'' options
b''|b'' b''|b'' b''|b''b''-b''b''-b'' patch-header
b''|b'' b''|b''b''-b''b''-b'' tests
b''|b'' b''|b'' b''|b''b''-b''b''-b'' control
b''|b'' b''|b''b''-b''b''-b'' upstream
b''|b'' b''|b'' b''|b''b''-b''b''-b'' metadata
b''|b'' b''|b''b''-b''b''-b'' watch
b''|b''b''-b''b''-b'' src
    b''|b''b''-b''b''-b'' hello.c

8 directories, 24 files
$ cat debian/patches/series
000-prefix-usr.patch
$ cat debian/patches/000-prefix-usr.patch
Description: set prefix=/usr patch
Author: Osamu Aoki <osamu@debian.org>
Index: debhello-0.0/Makefile
=====
--- debhello-0.0.orig/Makefile
+++ debhello-0.0/Makefile
@@ -1,4 +1,4 @@
-prefix = /usr/local
+prefix = /usr

all: src/hello

```

Here, **Makefile** in the upstream source tree doesn't need to be restored to the original state. The **dpkg-source** command invoked by the Debian packaging procedure in Section 4.8, understands the patch application state recorded by the **dquilt** program in the **.pc/** directory. As long as all the changes are committed by the **dquilt** command, the Debian source package can be built from the modified source tree.

Note



If the **.pc/** directory is missing, the **dpkg-source** command assumes that no patch was applied. That's why the more primitive patch generation methods like in Section 4.9.1 without generating the **.pc/** directory require the upstream source tree to be restored.

4.9.3 Patch durch dpkg-source --commit

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

Bearbeiten Sie die Quellen der Originalautoren.

```

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

```

Let's commit it.

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

Schauen Sie sich das Ergebnis an.

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

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

all: src/hello

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

8 directories, 24 files
```

Hier erledigt der Befehl **dpkg-source** genau das gleiche wie dies die Sequenz der **dquilt**-Befehle in Section 4.9.2 tat.

Chapter 5

Grundlagen

Für die grundlegenden Regeln der Debian-Paketierung wird hier ein breiter Überblick gegeben, der sich auf nicht-native Debian-Pakete im Format »3.0 (quilt)« konzentriert.

Note



Zu Gunsten der Klarheit werden einige Details bewusst übergangen. Bitte lesen Sie die Handbuchseiten von **dpkg-source(1)**, **dpkg-buildpackage(1)**, **dpkg(1)**, **dpkg-deb(1)**, **deb(5)** usw.

Das Debian-Quellpaket ist eine Gruppe von Eingabedateien (d.h. nicht eine einzelne Datei), die zum Bau des Debian-Binärpakets verwandt werden.

Das Debian-Binärpaket ist eine besondere Archivdatei, die die Gruppe der installierbaren Binärdaten mit ihren zugehörigen Informationen enthält.

Ein einzelnes Debian-Quellpaket kann mehrere Debian-Binärpakete, die in der Datei **debian/control** definiert sind, installieren.

Das nichtnative Debian-Paket im Format »3.0 (quilt)« ist das gängigste Debian-Quellpaketformat.

Note



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

5.1 Arbeitsablauf des Paketierens

Der Arbeitsablauf beim Paketieren, um ein Debian-Binärpaket zu erstellen, beinhaltet die Erstellung mehrere Dateien mit bestimmten Namen (siehe Section 5.4), wie dies in dem »Debian Policy Manual« definiert ist.

Stark vereinfacht kann der Arbeitsablauf beim Debian-Paketieren in folgende 10 Schritte zusammengefasst werden:

1. Der Tarball der Originalautoren wird als Datei *Paket-Version.tar.gz* heruntergeladen.
2. The upstream tarball is untarred to create many files under the *package-version/* directory.
3. The upstream tarball is copied (or symlinked) to the particular filename *packagename_version.orig.tar.gz*.
 - the character separating *package* and *version* is changed from - (hyphen) to _ (underscore)
 - **.orig** is added in the file extension.
4. The Debian package specification files are added to the upstream source under the *package-version/debian/* directory.

- Required specification files under the **debian/** directory:
 - debian/rules** Das ausführbare Skript zum Bau des Debian-Pakets (siehe Section 5.6)
 - debian/control** The package configuration file containing the source package name, the source build dependencies, the binary package name, the binary dependencies, etc. (see Section 5.7)
 - debian/changelog** The Debian package history file defining the upstream package version and the Debian revision in its first line (see Section 5.8)
 - debian/copyright** The copyright and license summary (see Section 5.9)
 - Optional specification files under the **debian/*** (see Section 5.13):
 - The **debmake** command invoked in the *package-version/* directory provides the initial set of these template configuration files.
 - Required specification files are generated even with the **-x0** option.
 - The **debmake** command does not overwrite any existing configuration files.
 - These files must be manually edited to their perfection according to the “Debian Policy Manual” and “Debian Developer’s Reference”.
5. The **dpkg-buildpackage** command (usually from its wrapper **debuild** or **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 “3.0 (quilt)” format using **dpkg-source(1)**
 - *package_version.orig.tar.gz* (copy or symlink of *package-version.tar.gz*)
 - *package_version-revision.debian.tar.xz* (tarball of *package-version/debian/**)
 - *package_version-revision.dsc*
 - Build the source using “**debian/rules build**” into **\$(DESTDIR)**
 - **DESTDIR=debian/binarypackage/** (single binary package)
 - **DESTDIR=debian/tmp/** (multi binary package)
 - Create the Debian binary package using **dpkg-deb(1)**, **dpkg-genbuildinfo(1)**, and **dpkg-genchanges(1)**.
 - *binarypackage_version-revision_arch.deb*
 - ... (There may be multiple Debian binary package files.)
 - *package_version-revision_arch.changes*
 - *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 *package_version-revision.dsc* and ‘*package_version-revision*_source.changes**’ files with the **deb-sign** command using your private GPG key.
10. Upload the set of the Debian source package files with the **dput** command to the Debian archive.

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.

Under some exceptional situation such as NEW uploads, uploads to the Debian archive may need to include Debian binary package files. For such situation, sign *package_version-revision_arch.changes* instead of ‘*package_version-revision*_source.changes**’ in the step 9, and upload the set of the Debian source and binary package files in the step 10.

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

See also [Source-only uploads](#).

Tip



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

Tip



There is very extensive documentation in [Chapter 6. Best Packaging Practices](#) in the “Debian Developer's Reference”. Please read it.

5.1.1 Das Paket debhelper

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

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

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

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

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

5.2 Historical perspective of Debian packaging practices

Let me oversimplify historical perspective of Debian packaging practices.

[Debian was started in 1990s](#) when upstream packages were available from public FTP sites such as [Sunsite](#). In those early days, Debian packaging used dpkg-source currently known as “Format : 1.0”:

- Upstream released packages in tar.gz format. They were obtained from some FTP sites.
- Debian applied “one big patch” as its source distribution and made its binary distribution form it.
- Many different approaches were adopted by different Debian developers to manage topic patches with custom scripts within the “one big patch”. — Chaos!

In order to address issues of old dpkg-source “Format : 1.0”, new dpkg-source “Format : 3.0 (quilt)” was invented around 2008:

- Upstream released packages in tar.gz/tar.bz2/tar.xz format optionally with signature files. They were obtained from some HTTP sites.
- Most Debian developers adopted patch queue mechanism of "Format: 3.0 (quilt)" to manage topic patches and made Debian packages while keeping files outside of `debian/` directory untouched.
- VCS, especially git, became popular and most Debian developers started to record this packaging practice in `gbp-buildpackage(1)` style.
- The `gbp-buildpackage(1)` workflow records the exact same content of the upstream tarball to VCS for source files outside of `debian/` directory (= patch-unapplied).

The use of Git repositories to distribute upstream packages with signed tags (supported feature since 2011) became very popular.

- The tarball centric `gbp-buildpackage(1)` workflow to record changes to VCS was cumbersome for some Debian developers and `dgit(1)` was invented in 2013.
- The Git centric `dgit-maint-debrebase(7)` and `dgit-maint-merge(7)` workflows to record changes to VCS are gaining popularity among these Debian developers.
- Source files recorded to VCS by `dgit-maint-debrebase(7)` and `git-maint-merge(7)` are modified upstream source files (= patch-applied).
- Uploaded source packages by `dgit-maint-debrebase(7)` and `git-maint-merge(7)` workflows still use `dpkg-source "Format: 3.0 (quilt)"`.

Debian also enforced the source-only upload when developing Debian/11 Bullseye (released in 2021).

In this tutorial, mostly simple tarball based `dpkg-source "Format: 3.0 (quilt)"` examples are presented as an introductory purpose.

Please assess these VCS usage approaches by yourself later to decide which one to deploy as your preferred one.

5.3 Future perspective on Debian packaging practices

Please look around to understand how Debian packaging practices are evolving and follow the current general trends if possible.

- [DEP - Debian Enhancement Proposals](#)
 - Debian Enhancement Proposals (or DEPs, for short) offer a device to organize discussions about various kinds of enhancements in the Debian project, reflect their current status and, in particular, archive their outcomes.
- [Debian Trends](#)
 - This page provides some historical perspective about how those practices evolved.
- [Debian git packaging maintainer branch formats and workflows](#)
 - Nice packaging summary by the latest `dgit` tool providers.

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

- [Debian Sources](#) — code search tool
 - [Debian Code Search](#) — wiki page describing its usage
- [Debian Code Search](#) — another code search tool

5.4 Paketname und -version

Falls die Quellen der Originalautoren als **hello-0.9.12.tar.gz** vorliegen, können Sie **hello** als Namen des Quellpakets der Originalautoren und **0.9.12** als Version der Originalautoren nehmen.

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

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

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

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

See the exact definition in [Chapter 5 - Control files and their fields](#) in the “Debian Policy Manual”.

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

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

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

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

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

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

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

The version comparison rule can be summarized as:

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

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

```
0.0 < 0.5 < 0.10 < 0.99 < 1 < 1.0~rc1 < 1.0 < 1.0+b1 < 1.0+nmU1 < 1.1 < 2.0
```

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

¹For 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.

5.5 Natives Debian-Paket

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

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

Caution



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

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

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

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

Tip



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

^aUse of the “**debmake -t ...**” command or “**git deborig -f HEAD**” can help this workflow. See Section 6.2 and **dggit-maint-merge(7)**.

Tip



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

5.6 debian/rules

Das Skript **debian/rules** ist das ausführbare Skript, um das Debian-Paket zu bauen.

- The **debian/rules** script re-targets the upstream build system (see Section 5.18) to install files in the **\$(DESTDIR)** and creates the archive file of the generated files as the **deb** file.
 - The **deb** file is used for the binary distribution and installed to the system using the **dpkg** command.
- The **dh** command is normally used as the front-end to the build system inside the **debian/rules** script.
- **\$(DESTDIR)** path depends on the build type.

- **\$(DESTDIR)=debian/binarypackage** (single binary package)
- **\$(DESTDIR)=debian/tmp** (multiple binary package)

5.6.1 dh

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

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

Note



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

5.6.2 Einfache debian/rules

Thanks to this abstraction of the **dh** command², the Debian policy compliant **debian/rules** file supporting all the required targets can be written as simple as³:

Simple debian/rules:

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

%:
    dh $@
```

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

Tip



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

²This simplicity is available since version 7 of the **debhelper** package. This guide assumes the use of **debhelper** version 13 or newer.

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

5.6.3 Angepasste debian/rules

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

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

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

- **configure**,
- **Makefile**,
- **setup.py** oder
- **Build.PL**.

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

Tip



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

The **debmake** command creates the initial template file taking advantage of the above simple **debian/rules** file example while adding some extra customizations for package hardening, etc. You need to know how underlying build systems work under the hood (see Section 5.18) to address their irregularities using package customization.

- See Section 4.7 for basic customization of the template **debian/rules** file generated by the **debmake** command.
- See Section 5.22 for multiarch customization.
- See Section 5.23 for hardening customization.

5.6.4 Variablen für debian/rules

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

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

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

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

buildflags.mk Setzt die Bauschalter **CFLAGS**, **CPPFLAGS**, **CXXFLAGS**, **OBJCFLAGS**, **OBJCXXFLAGS**, **GCJFLAGS**, **FFLAGS**, **FCFLAGS** und **LDFLAGS**.

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

Beispielsweise können Sie eine Zusatzoption zu **CONFIGURE_FLAGS** für die Zielarchitektur **linux-any** durch Hinzufügen von folgendem zur **debian/rules** hinzufügen:

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

Tip



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

Siehe Section 5.22, **dpkg-architecture**(1) und **dpkg-buildflags**(1).

5.6.5 Reproduzierbares Bauen

Here are some recommendations to attain a reproducible build result.

- Betten Sie keine Zeitstempel, basierend auf der Systemzeit, ein.
- Verwenden Sie »**dh \$@**« in der **debian/rules**, um auf die neusten Funktionalitäten von **debhelper** zuzugreifen.
- Exportieren Sie die Bauumgebung als »**LC_ALL=C.UTF-8**« (siehe Section 7.15).
- Set the timestamp used in the upstream source from the value of the debhelper-provided environment variable **\$SOURCE_DATE_EPOCH**.
- Lesen Sie mehr unter [Reproduzierbares Bauen](#).
 - [Reproduzierbares Bauen - Wie geht es.](#)
 - [Reproduzierbares Bauen - Zeitstempelvorschlag.](#)

Die durch **dpkg-genbuildinfo**(1) erstellte Steuerdatei **source-name_source-version_arch.buildinfo** zeichnet die Bauumgebung auf. Siehe **deb-buildinfo**(5)

5.7 debian/control

Die Datei **debian/control** besteht aus Metadatenblöcken, die durch Leerzeilen getrennt sind. Jeder Metadatenblock definiert folgendes in dieser Reihenfolge:

- Metadaten für das Debian-Quellpaket
- Metadaten für das Debian-Binärpaket

See [Chapter 5 - Control files and their fields](#) of the “Debian Policy Manual” for the definition of each meta data.

Note



The **debmake** command sets the **debian/control** file with “**Build-Depends: debhelper-compat (= 13)**” to set the **debhelper** compatibility level.

Tip



If an existing package has lower than **debhelper** compatibility level 9, probably it's time to update its packaging.

5.7.1 Aufteilung eines Debian-Binärpakets

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.

Bitte prüfen Sie die Beispiele in dieser Anleitung:

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

5.7.1.1 debmake -b

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

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

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

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

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

- **bin**: C/C++ compiled ELF binary code package (any, foreign) (default, alias: "", i.e., *null-string*)
- **data**: Data (fonts, graphics, ...) package (all, foreign) (alias: **da**)
- **dev**: Library development package (any, same) (alias: **de**)
- **doc**: Documentation package (all, foreign) (alias: **do**)
- **lib**: Library package (any, same) (alias: **l**)
- **perl**: Perl script package (all, foreign) (alias: **pl**)
- **python3**: Python (version 3) script package (all, foreign) (alias: **py3**)
- **ruby**: Ruby script package (all, foreign) (alias: **rb**)
- **nodejs**: Node.js based JavaScript package (all, foreign) (alias: **js**)
- **script**: Shell script package (all, foreign) (alias: **sh**)

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

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

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

5.7.1.2 Pakettrennungsszenarien und -beispiele

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

- eine Bibliotheksquelle *libfoo-1.0.tar.gz*
- eine Werkzeugquelle *bar-1.0.tar.gz*, geschrieben in einer kompilierten Sprache
- eine Werkzeugquelle *baz-1.0.tar.gz*, geschrieben in einer interpretierten Sprache

<i>Binärpaket</i>	<i>Typ</i>	Architecture:	Multi-Arch:	Paketinhalt
libfoo1	lib*	any	same	die Laufzeitbibliothek, koinstallierbar
libfoo-dev	dev*	any	same	the shared library header files etc., co-installable
libfoo-tools	bin*	any	foreign	the run-time support programs, not co-installable
libfoo-doc	doc*	all	foreign	the shared library documentation files
<i>bar</i>	bin*	any	foreign	the compiled program files, not co-installable
<i>bar-doc</i>	doc*	all	foreign	the documentation files for the program
<i>baz</i>	script	all	foreign	the interpreted program files

5.7.1.3 Der Bibliothekspaketname

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

The binary library package must be renamed from **libfoo7** to **libfoo8** to keep the **unstable** suite system working for all dependent packages after the upload of the package based on the **libfoo-8.0.tar.gz**.

Warning



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

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

- Use the **unversioned -dev** package name: **libfoo-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 **libfoo7** to **libfoo8** to prevent dependency breakage in the **unstable** archive during the library transition.
 - This approach should be used if the simple binNMU resolves the library dependency quickly for all affected packages. (**ABI** change by dropping the deprecated **API** while keeping the active API unchanged.)
 - This approach may still be a good idea if manual code updates, etc. can be coordinated and manageable within limited packages. (**API** change)
- Use the **versioned -dev** package names: **libfoo7-dev** and **libfoo8-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 **libfoo-dev**.
 - * Make both **libfoo7-dev** and **libfoo8-dev** provide **libfoo-dev**.

- * The source package needs to be renamed as **libfoo7-7.0.tar.gz** and **libfoo8-8.0.tar.gz** respectively from **libfoo-?.0.tar.gz**.
- * The package specific install file path including **libfoo7** and **libfoo8** respectively for header files etc. needs to be chosen to make them co-installable.
- Do not use this heavy handed approach, if possible.
- This approach should be used if the update of multiple dependent packages require manual code updates, etc. (API change) Otherwise, the affected packages become RC buggy with FTBFS.

Tip

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

Siehe Section [5.20](#).

5.7.2 Substvar

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

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

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

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

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

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

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

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

5.7.3 binNMU-sicher

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~**)

5.8 debian/changelog

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

- Even if you are uploading your package by yourself, you must document all non-trivial user-visible changes such as:
 - sicherheitsrelevante Änderungen
 - Änderungen in der Benutzerschnittstelle
- If you are asking your sponsor to upload it, you should document changes more comprehensively, including all packaging related ones, to help reviewing your package.
 - The sponsor shouldn’t second guess your thought behind your package.
 - Die Zeit des Sponsors ist wertvoller als Ihre.

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

Der Befehl **debchange** (auch als **dch** bekannt) wird typischerweise zur Bearbeitung dieser Datei verwendet.

Tip



Sie können die Datei **debian/changelog** händisch mit einem beliebigen Texteditor bearbeiten, solange Sie den von **debchange** verwandten Formatierungskonventionen folgen.

Tip



Die in der Datei **debian/changelog** verwandte Datumszeichenkette kann mit dem Befehl »**LC_ALL=C date -R**« manuell erstellt werden.

Dieser wird durch den Befehl **dh_installchangelogs** als **changelog.Debian.gz** in das Verzeichnis **/usr/share/doc/Binärpaket** installiert.

Der Changelog der Originalautoren wird als **changelog.gz** in das Verzeichnis **/usr/share/doc/Binärpaket** installiert.

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

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

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

5.9 debian/copyright

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

Sie sollten es als [maschinenlesbare Datei debian/copyright](#) (DEP-5) formatieren.

Caution



Die Datei **debian/copyright** sollte sortiert sein, um die generischen Dateimuster am Anfang der Liste zu behalten. Siehe Section [6.4](#).

Der Befehl **debmake** erstellt die anfängliche, DEP-5-kompatible Vorlagendatei, indem er den gesamten Quellbaum überprüft. Er verwendet einen internen Lizenzprüfer, um jeden Lizenztext auszuwerten.⁵

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

Note



Falls Sie Probleme mit diesem Lizenzprüfer feststellen, reichen Sie bitten gegen das Paket **debmake** einen Fehlerbericht mit dem problematischen Teil des Textes, der das Copyright und die Lizenz enthält, ein.

Note



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

Tip



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

5.10 debian/patches/*

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

Note



Das native Debian-Paket (siehe Section [5.5](#)) verwendet diese Dateien nicht.

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

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

- Der Befehl **diff**
 - Siehe Section [4.9.1](#)
 - Primitive but versatile method
 - * Patches may come from other distros, mailing list postings, or cherry-picked patches from the upstream **git** repository with the “**git format-patches**” command
 - Missing the **.pc/** directory
 - Unmodified upstream source tree
 - Manually update the **debian/patches/series** file
- Der Befehl **dquilt**
 - Siehe Section [3.4](#)
 - Basic convenient method
 - Proper generation of the **.pc/** directory data
 - Modified upstream source tree
- Der Befehl »**dpkg-source --commit**«
 - Siehe Section [4.9.3](#)
 - Newer elegant method
 - Proper generation of the **.pc/** directory data
 - Modified upstream source tree
- The automatic patch generation by the **dpkg-buildpackage**
 - Siehe Section [5.16](#)
 - Add **single-debian-patch** in the **debian/source/local-options** file
 - Set the **debian/source/local-patch-header** file
 - Missing the **.pc/** directory
 - Modified upstream source tree in the Debian branch (**master**)
- Der Befehl **gbp-pq**
 - basic **git** work flow with the **git-buildpackage** package
 - Missing the **.pc/** directory
 - Modified upstream source tree in the throw-away branch (**patch-queue/master**)
 - Unmodified upstream source tree in the Debian branch (**master**)
- Der Befehl **gbp-dpm**
 - more elaborate **git** work flow with the **git-dpm** package
 - Missing the **.pc/** directory
 - Modified upstream source tree in the patched branch (**patched/whatever**)
 - Unmodified upstream source tree in the Debian branch (**master/whatever**)

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

Tip



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

5.10.1 dpkg-source -x

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

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

Falls Sie den Quellbaum unverändert belassen wollen (beispielsweise für die Verwendung in Section 5.15), verwenden Sie bitte die Option **--skip-patches**.

5.10.2 dquilt und dpkg-source

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

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

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

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

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

5.11 debian/upstream/signing-key.asc

Some packages are signed by a GPG key.

For example, [GNU hello](https://ftp.gnu.org/gnu/hello/) can be downloaded via HTTP from <https://ftp.gnu.org/gnu/hello/> . There are sets of files:

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

Let’s pick the latest version set.

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

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

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

Tip



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

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

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

Setzen Sie dann die entsprechende Datei **debian/watch** wie folgt:

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

Jetzt wird der Befehl **uscan** die Echtheit des Pakets mittels GPG-Signaturen prüfen.

5.12 debian/watch und DFSG

Debian nimmt die Freiheit von Software ernst und folgt der [DFSG](#).

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

- List the files to be removed in the **Files-Excluded** stanza of the **debian/copyright** file.
- List the URL to download the upstream tarball in the **debian/watch** file.
- Führen Sie den Befehl **uscan** aus, um einen neuen Tarball der Originalautoren herunterzuladen.
 - Verwenden Sie alternativ den Befehl »**gbp import-orig --uscan --pristine-tar**«.
- Der resultierende Tarball hat die Versionsnummer mit einer zusätzlichen Endung **+dfsg**.

5.13 Weitere debian/*-Dateien

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

Tip



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

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

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

Here is the alphabetical list of notable optional configuration files.

Binärpaket.bug-control ^{-x3} installed as **usr/share/bug/binarypackage/control** in *binarypackage*. See Section [5.27](#).

Binärpaket.bug-presubj ^{-x3} installed as **usr/share/bug/binarypackage/presubj** in *binarypackage*. See Section [5.27](#).

Binärpaket.bug-script ^{-x3} installed as **usr/share/bug/binarypackage** or **usr/share/bug/binarypackage/script** in *binarypackage*. See Section [5.27](#).

Binärpaket.bash-completion List bash completion scripts to be installed.

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

Siehe `dh_bash-completion(1)`.

clean ^{-x2} List files that should be removed but are not cleaned by the `dh_auto_clean` command.

Siehe `dh_auto_clean(1)` und `dh_clean(1)`.

compat ^{-x3} Previously, this set the `debhelper` compatibility level.

Now, use `Build-Depends: debhelper-compat (= 13)` in `debian/control` to specify the compatibility level.

See “COMPATIBILITY LEVELS” in `debhelper(8)`.

Binärpaket.conf No need for this file now since all files in the `etc/` directory are conffiles for recent “compat ≥ 3 ”.

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.

Siehe `dh_installdeb(1)`.

Binärpaket.config This is the `debconf config` script used for asking any questions necessary to configure the package. See Section 5.21.

Binärpaket.cron.hourly ^{-x3} Installed into the `etc/cron/hourly/binarypackage` file in *binarypackage*.

Siehe `dh_installcron(1)` und `cron(8)`.

Binärpaket.cron.daily ^{-x3} Installed into the `etc/cron/daily/binarypackage` file in *binarypackage*.

Siehe `dh_installcron(1)` und `cron(8)`.

Binärpaket.cron.weekly ^{-x3} Installed into the `etc/cron/weekly/binarypackage` file in *binarypackage*.

Siehe `dh_installcron(1)` und `cron(8)`.

Binärpaket.cron.monthly ^{-x3} Installed into the `etc/cron/monthly/binarypackage` file in *binarypackage*.

Siehe `dh_installcron(1)` und `cron(8)`.

Binärpaket.cron.d ^{-x3} Installed into the `etc/cron.d/binarypackage` file in *binarypackage*.

Siehe `dh_installcron(1)`, `cron(8)` und `crontab(5)`.

Binärpaket.default ^{-x3} If this exists, it is installed into `etc/default/binarypackage` in *binarypackage*.

Siehe `dh_installinit(1)`.

Binärpaket.dirs ^{-x3} List directories to be created in *binarypackage*.

Siehe `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.

Binärpaket.doc-base ^{-x2} Installed as the `doc-base` control file in *binarypackage*.

See `dh_installdocs(1)` und [Debian doc-base Manual](#) provided by the `doc-base` package.

Binärpaket.docs ^{-x2} List documentation files to be installed in *binarypackage*.

Siehe `dh_installdocs(1)`.

Binärpaket.emacsen-compat ^{-x3} Installed into `usr/lib/emacsen-common/packages/compat/binarypackage` in *binarypackage*.

Siehe `dh_installemacsen(1)`.

Binärpaket.emacsen-install ^{-x3} Installed into **usr/lib/emacsen-common/packages/install/binarypackage** in *binarypackage*.

Siehe **dh_installemacsen(1)**.

Binärpaket.emacsen-remove ^{-x3} Installed into **usr/lib/emacsen-common/packages/remove/binarypackage** in *binarypackage*.

Siehe **dh_installemacsen(1)**.

Binärpaket.emacsen-startup ^{-x3} Installed into **usr/lib/emacsen-common/packages/startup/binarypackage** in *binarypackage*.

Siehe **dh_installemacsen(1)**.

Binärpaket.examples ^{-x2} List example files or directories to be installed into **usr/share/doc/binarypackage/examples/** in *binarypackage*.

Siehe **dh_installexamples(1)**.

gbp.conf Falls diese existiert, funktioniert sie als Konfigurationsdatei für den Befehl **gbp**.

Siehe **gbp.conf(5)**, **gbp(1)** und **git-buildpackage(1)**.

Binärpaket.info ^{-x2} Führt Info-Dateien auf, die in *Binärpaket* installiert werden.

Siehe **dh_installinfo(1)**.

Binärpaket.init ^{-x3} Installiert in **etc/init.d/Binärpaket** in *Binärpaket*.

Siehe **dh_installinit(1)**.

Binärpaket.install ^{-x2} Liste der Dateien, die installiert werden sollten, aber nicht durch den Befehl **dh_auto_install** installiert wurden.

Siehe **dh_install(1)** und **dh_auto_install(1)**.

license-examples/* ^{-x4} Dies sind durch den Befehl **debmake** erstellte Copyright-Datei-Beispiele. Verwenden Sie diese als Referenz zur Erstellung der Datei **copyright**.

Bitte sorgen Sie dafür, diese Dateien zu löschen.

Binärpaket.links ^{-x2} List pairs of source and destination files to be symlinked. Each pair should be put on its own line, with the source and destination separated by whitespace.

Siehe **dh_link(1)**.

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

Siehe **dh_lintian(1)**, **lintian(1)** und [Lintian User's Manual](#).

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

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

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

Binärpaket.manpages ^{-x2} List man pages to be installed.

Siehe **dh_installman(1)**.

Binärpaket.menu (veraltet, nicht mehr installiert) [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**.

Siehe **dh_installchangelogs(1)**.

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

Siehe **dpkg-source**(1), Section 3.4 und Section 4.9.

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

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

Binärpaket.preinst ^{-x2}, **Binärpaket.postinst** ^{-x2}, **Binärpaket.prerm** ^{-x2}, **Binärpaket.postrm** ^{-x2} These maintainer scripts are installed into the **DEBIAN** directory.

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

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

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

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

Siehe **dh_installdocs**(1).

This file provides the information specific to the Debian package.

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

Siehe **dh_systemd_enable**(1), **dh_systemd_start**(1) und **dh_installinit**(1).

source/format ^{-x1} The Debian package format.

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

Siehe »QUELLPAKET-FORMATE« in **dpkg-source**(1).

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

Siehe **dh_lintian**(1) und **lintian**(1).

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

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

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

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

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

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

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

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

Siehe **dh_makeshlibs**(1) und Section 5.20.1..

Binärpaket.templates This is the **debconf templates** file used for asking any questions necessary to configure the package. See Section 5.21.

tests/control This is the RFC822-style test meta data file defined in DEP-8. See **autopkgtest**(1) and Section 5.24.

TODO Installed into the first binary package listed in the **debian/control** file as **usr/share/doc/binarypackage/TODO.Debian**.
 Siehe **dh_installdocs(1)**.

Binärpaket.tmpfile ^{-x3} If this exists, it is installed into **usr/lib/tmpfiles.d/binarypackage.conf** in *binarypackage*.
 Siehe **dh_systemd_enable(1)**, **dh_systemd_start(1)** und **dh_installinit(1)**.

Binärpaket.upstart ^{-x3} If this exists, it is installed into **etc/init/package.conf** in the package build directory. (deprecated)
 Siehe **dh_installinit(1)** und Section 8.1.

watch ^{-x1} The control file for the **uscan** command to download the latest upstream version.
 This control file may be configured to verify the authenticity of the tarball using its GPG signature (see Section 5.11).
 Siehe Section 5.12 und **uscan(1)**.

Here are a few reminders for the above list.

- For a single binary package, the *binarypackage.* part of the filename in the list may be removed.
- For a multi binary package, a configuration file missing the *binarypackage.* part of the filename is applied to the first binary package listed in the **debian/control**.
- When there are many binary packages, their configurations can be specified independently by prefixing their name to their configuration filenames such as *package-1.install*, *package-2.install*, etc.
- Some template configuration files may not be created by the **debmake** command. In such cases, you need to create them with an editor.
- Unusual configuration template files generated by the **debmake** command with an extra **.ex** suffix need to be activated by removing that suffix.
- Nicht verwendete Vorlagendateien, die durch den Befehl **debmake** erstellt wurden, sollten entfernt werden.
- Kopieren Sie die Konfigurationsvorlagendateien wie benötigt nach den Dateinamen, die ihren Programm-paketnamen entsprechen.

5.14 Anpassung der Debian-Paketierung

Let's recap the customization of the Debian packaging.

All customization data for the Debian package resides in the **debian/** directory. A simple example is given in Section 4.7. Normally, this customization involves a combination of the following:

- The Debian package build system can be customized through the **debian/rules** file (see Section 5.6.3).
- The Debian package installation path etc. can be customized through the addition of configuration files such as *package.install* and *package.docs* in the **debian/** directory for the **dh_*** commands from the **debhelper** package (see Section 5.13).

When these are not sufficient to make a good Debian package, modifications to the upstream source recorded as the **-p1** patches in the **debian/patches/** directory is deployed. These patches are applied in the sequence defined in the **debian/patches/series** file before building the package (see Section 5.10). Simple examples are given in Section 4.9.

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

Note



Schicken Sie den Patch, der das eigentliche Problem adressiert, an die Originalautoren, falls er für die Originalautoren nützlich ist.

5.15 Aufzeichnen im VCS (standard)

Typischerweise wird [Git](#) als **VCS** benutzt, um die Debian-Paketieraktivitäten mit den nachfolgenden Verzweigungen aufzuzeichnen.

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

Tip



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

Tip



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

Tip



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

5.16 Aufzeichnen im VCS (Alternative)

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

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

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

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

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

Tip



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

Tip



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

5.17 Pakete ohne irrelevante Inhalte bauen

Es gibt ein paar Fälle, die zur Aufnahme von ungewünschten Inhalten in das erstellte Debian-Quellpaket führen.

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

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

Es gibt mehrere Methoden, die Aufnahme ungewünschter Inhalte zu vermeiden

5.17.1 Korrektur durch **debian/rules clean**

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

Note



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

5.17.2 Korrektur mittels VCS

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

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

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

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

Tip



Falls der Quellbaum nicht in einem VCS verwaltet wird, sollten Sie vor dem ersten Bau »**git init**; **git add -A**.; **git commit** ausführen.

5.17.3 Korrektur durch extend-diff-ignore

Dies ist für nichtnative Pakete.

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

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

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

Note



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

Tip



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

5.17.4 Korrektur durch tar-ignore

Dies ist für native Pakete.

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

Note



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

5.18 Bausysteme der Originalautoren

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

Three typical build systems are described here. The situation of other build systems are very similar to these since `debhelper(7)` does most of the work and helps you build a Debian package.

Tip



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

5.18.1 Autotools

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

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

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

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

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

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

Falls Sie mehr über die Autotools lernen möchten, lesen Sie:

- [GNU-Automake-Dokumentation](#)
- [GNU-Autoconf-Dokumentation](#)
- [Autotools-Anleitung](#)
- [Einführung in die Autotools \(Autoconf, Automake und Libtool\)](#)
- [Autotools Mythbuster](#)

5.18.2 CMake

CMake hat 4 Schritte.

1. Einrichten des Bausystems (“**vim CMakeLists.txt config.h.in**”)
2. Konfiguration des Bausystems (“**cmake**”)
3. build the source tree (“**make**”)
4. install the binary files (“**make install**”)

Der Tarball der Originalautoren enthält keine automatisch erstellten Dateien und wird durch den Befehl **tar** nach Schritt 1 erstellt.

Der Paketbetreuer muss sich um die Schritte 2 bis 4 kümmern.

Falls Sie mehr über CMake lernen möchten, lesen Sie bitte:

- [CMake](#)
- [CMake-Anleitung](#)

5.18.3 Meson

Meson has 4 steps.

1. set up the **Meson** build system (“**vim meson.build**”)
2. set up the **Ninja** build system and configure it (“**meson setup builddir && cd builddir**” and “**meson configure**”)
3. build the source tree (“**meson compile**”)
4. install the binary files (“**meson install**”)

Der Tarball der Originalautoren enthält keine automatisch erstellten Dateien und wird durch den Befehl **tar** nach Schritt 1 erstellt.

Der Paketbetreuer muss sich um die Schritte 2 bis 4 kümmern.

If you wish to learn more on the Meson, please see:

- [The Meson Build system](#)
- [Ninja](#)

5.18.4 Python Distutils

Python Distutils hat 3 Schritte.

1. Einrichten und Konfigurieren des Bausystems (“**vim setup.py**”)
2. build the source tree (“**python3 setup.py build**”)
3. install the binary files (“**python3 setup.py install**”)

The upstream maintainer usually performs step 1 and builds the upstream tarball for distribution using the “**python3 setup.py sdist**” command.

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

These days, most upstream maintainers of Python packages use setuptools with wheel. Since setuptools is an extension of distutils, this step 2 works as expected even if **setup.py** doesn’t explicitly uses distutils.

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

- [Python3](#)
- [Distutils](#)
- [setuptools](#)

5.19 Fehlersuchinformationen

Das Debian-Paket wird mit den Fehlersuchinformationen gebaut, aber in das Binärpaket nach entfernen der Fehlersuchinformationen paketierte, wie von [Kapitel 10 - Dateien](#) des »Debian Policy Manual« verlangt.

Siehe

- [6.7.9. Best practices for debug packages](#) der »Debian-Entwicklerreferenz«.
- [18.2 Fehlersuchinformation in separaten Dateien](#) des ”Debugging with gdb”
- **dh_strip(1)**
- **strip(1)**
- **readelf(1)**
- **objcopy(1)**
- Debian wiki [DebugPackage](#)
- Debian wiki [AutomaticDebugPackages](#)
- Debian debian-devel post: [Status on automatic debug packages](#) (2015-08-15)

5.19.1 Neue -dbgsym-Pakete (Stretch 9.0 und danach)

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

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

- The **debian/rules** file shouldn’t explicitly contain **dh_strip**.
- Remove **debian/compat**.
- Set the **Build-Depends** to **debhelper-compat (>=13)** while removing **Build-Depends** to **debhelper** in **debian/control**.

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

- Drop definition entries of such **-dbg** packages in the **debian/control** file.
- Replace **”dh_strip --dbg-package=package”** with **”dh_strip --dbgsym-migration=package”** in the **debian/rules** file to avoid file conflicts with the (now obsolete) **-dbg** package. See **dh_strip(1)**.
- Remove **debian/compat**.
- Set the **Build-Depends** to **debhelper-compat (>=13)** while removing **Build-Depends** to **debhelper** in **debian/control**.

5.20 Bibliothekspaket

Die Paketierung von Bibliothekssoftware verlangt von Ihnen deutlich mehr Arbeit als normal. Es folgen ein paar Erinnerungen zur Paketierung von Bibliothekssoftware:

- The library binary package must be named as in Section [5.7.1.3](#).
- Debian ships shared libraries such as **/usr/lib/<triplet>/libfoo-0.1.so.1.0.0** (see Section [5.22](#)).
- Debian encourages using versioned symbols in the shared library (see Section [5.20.1](#)).
- Debian doesn’t ship ***.la** libtool library archive files.
- Debian discourages using and shipping ***.a** static library files.

Before packaging shared library software, see:

- [Chapter 8 - Shared libraries](#) of the “Debian Policy Manual”
- [10.2 Libraries](#) of the “Debian Policy Manual”
- [6.7.2. Libraries](#) of the “Debian Developer’s Reference”

For the historic background study, see:

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

5.20.1 Bibliothekssymbole

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

An oversimplified method for the library packaging is as follows.

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

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

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

Sie sollten auch Folgendes prüfen:

- Debian wiki [UsingSymbolsFiles](#)
- Debian wiki [Projects/ImprovedDpkgShlibdeps](#)
- Debian-KDE-Team [Working with symbols files](#)

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

⁷The 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.

- Section [8.11](#)
- Section [8.12](#)

Tip



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

5.20.2 Bibliotheksübergänge

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

Release team has the [transition tracker](#). See [Transitions](#).

Caution



Please make sure to rename binary packages as in Section [5.7.1.3](#).

5.21 Debconf

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

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

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

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

See **dh_installdebconf(1)**, **debconf(7)**, **debconf-devel(7)** and [3.9.1 Prompting in maintainer scripts](#) in the “Debian Policy Manual”.

5.22 Multiarch

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

You should read the following references in detail.

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

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

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

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

See:

- [Section 5.6.4](#)
- **dpkg-architecture(1)**
- [Section 5.7.1.1](#)
- [Section 5.7.1.2](#)

5.22.1 The multiarch library path

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

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

Table 5.1 The multiarch library path options

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

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

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

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

- Replace all occurrences of `/usr/lib/` with `/usr/lib/*/` in `debian/foo.install` files.

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

Note



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

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

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

5.22.2 The multiarch header file path

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

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

Table 5.2 The multiarch header file path options

Classic path	i386 multiarch path	amd64 multiarch path
<code>/usr/include/</code>	<code>/usr/include/i386-linux-gnu/</code>	<code>/usr/include/x86_64-linux-gnu/</code>
<code>/usr/include/package-name/</code>	<code>/usr/include/i386-linux-gnu/package-name/</code>	<code>/usr/include/x86_64-linux-gnu/package-name/</code>
	<code>/usr/lib/i386-linux-gnu/package-name/</code>	<code>/usr/lib/x86_64-linux-gnu/package-name/</code>

The use of the `/usr/lib/<triplet>/package-name/` path for the library files allows the upstream maintainer to use the same install script for the multiarch system with `/usr/lib/<triplet>` and the biarch system with `/usr/lib<qual>/`.⁸

The use of the file path containing `package-name` enables having more than 2 development libraries simultaneously installed on a system.

5.22.3 The multiarch *.pc file path

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

Table 5.3 The `*.pc` file path options

Classic path	i386 multiarch path	amd64 multiarch path
<code>/usr/lib/pkgconfig/</code>	<code>/usr/lib/i386-linux-gnu/pkgconfig/</code>	<code>/usr/lib/x86_64-linux-gnu/pkgconfig/</code>

⁸This 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."

5.23 Compiler hardening

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

You should read the following references in detail.

- Debian wiki [Hardening](#)
- Debian wiki [Hardening Walkthrough](#)

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

5.24 Continuous integration

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

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

See documents in the **/usr/share/doc/autopkgtest/** directory and [4. autopkgtest: Automatic testing for packages](#) of the “Ubuntu Packaging Guide”.

Note



Testing of the binary packages during their building time can be accommodated by **dh_auto_test**.

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

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

5.25 Other new requirements

Debian packaging practices are moving target. Please keep your eyes on [DEP - Debian Enhancement Proposals](#).

5.26 Bootstrapping

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

Tip



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

5.27 Bug reports

The **reportbug** command used for the bug report of *binarypackage* can be customized by the files in **usr/share/bug/binarypackage**.

The **dh_bugfiles** command installs these files from the template files in the **debian/** directory.

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

Siehe **dh_bugfiles(1)** und [reportbug's Features for Developers](#)

Tip



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

Chapter 6

Debmake-Optionen

Es gibt einige bemerkenswerte Optionen für den Befehl **debmake**.

6.1 Abkürzungs-Optionen (-a, -i)

Der Befehl **debmake** bietet 2 Abkürzungsoptionen.

- **-a**: öffnen des Tarballs der Originalautoren
- **-i**: Ausführen der Skripte zum Bau des Binärpakets

Das Beispiel in obigem Chapter 4 kann so einfach wie folgt sein.

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

Tip



Eine URL wie »<https://www.example.org/DL/Paket-1.0.tar.gz>« kann für die Option **-a** verwandt werden.

Tip



Eine URL wie »<https://arm.koji.fedoraproject.org/packages/ibus/1.5.7/3.fc21/src/ibus-1.5.7-3.fc21.src.rpm>« kann auch für die Option **-a** verwandt werden.

6.1.1 Python-Modul

Sie können direkt aus dem als Tarball angebotenen Python-Paket *Pythonmodule-1.0.tar.gz* ein funktionierendes Debian-Paket mit einem einzelnen Programm und einer vernünftigen Paketbeschreibung erstellen. Es müssen die Option **-b** für das Festlegen des Pakettyps **python** und die Option **-s** für das Kopieren der Paketbeschreibung aus dem Paket der Originalautoren festgelegt werden.

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

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

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

6.2 Snapshot upstream tarball (-d, -t)

This test building scheme is good for the git repository organized as described in **gbp-buildpackage**(7) which uses the master, upstream, and pristine-tar branches.

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

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

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

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

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

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

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

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

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

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

6.3 debmake -cc

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

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

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

6.4 debmake -k

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

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

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

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

Tip



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

6.5 debmake -j

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

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

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

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

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

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

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

```
$ debuild
```

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

Note



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

6.6 debmake -x

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

- See Section [8.1](#) for cherry-picking of the template files.

Note



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

6.7 debmake -P

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

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

6.8 debmake -T

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

Chapter 7

Tipps

Es folgen einige merkwürdige Tipps zur Debian-Paketierung.

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 binary upload

7.1 Debdiff

Sie können die Dateiinhalte in zwei Quellpaketen mit dem Befehl **debdiff** vergleichen.

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

Sie können auch Dateilisten in zwei Gruppen von Debian-Binärpaketen mit dem Befehl **debdiff** vergleichen.

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

Diese sind nützlich, um herauszufinden, was sich in Quellpaketen geändert hat und zu prüfen, ob unbeabsichtigte Änderungen beim Aktualisieren der Binärpakete erfolgten, wie beispielsweise falsch angeordnete oder entfernte Dateien.

7.2 Dget

Mit dem Befehl **dget** können Sie eine Gruppe von Dateien für das Debian-Quellpaket herunterladen.

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

7.3 Debc

Sie sollten erstellte Pakete mit dem Befehl **debc** lokal installieren, um sie zu testen.

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

7.4 Piuparts

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

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

Note

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

7.5 debsign

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

For uploading source-only package (normal):

```
$ debsign package_version-rev_source.changes
```

For uploading binary package:

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

7.6 dput

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

For uploading source-only package (normal):

```
$ dput package_version-rev_source.changes
```

For uploading binary package:

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

7.7 bts

After uploading the package, you will receive bug reports. It is an important duty of a package maintainer to manage these bugs properly as described in [5.8. Handling bugs](#) of the “Debian Developer’s Reference”.

The **bts** command is a handy tool to manage bugs on the [Debian Bug Tracking System](#).

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

7.8 git-buildpackage

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

- **gbp import-dsc**: import the previous Debian source package to the git repository.
- **gbp import-orig**: import the new upstream tar to the git repository.
 - The **--pristine-tar** option for the **git import-orig** command enables storing the upstream tarball in the same git repository.
 - The **--uscan** option as the last argument of the **gbp import-orig** command enables downloading and committing the new upstream tarball into the git repository.
- **gbp dch**: generate the Debian changelog from the git commit messages.
- **gbp buildpackage**: build the Debian binary package from the git repository.
 - The **sbuid** package can be used as its clean chroot build backend either by configuration or adding **--git-builder='sbuid -A -s --source-only-changes -v -d unstable'**

- **gbp pull**: update the **debian**, **upstream** and **pristine-tar** branches safely from the remote repository.
- The **gbp pq**, **git-dpm** or **quilt** (or alias **dquilt**) commands are used to manage quilt patches.
 - The **dquilt** command is the simplest to learn and requires you to commit the resulting files manually with the **git** command to the **master** branch.
 - The “**gbp pq**” command provides the equivalent functionality of patch set management without using **dquilt** and eases including upstream git repository changes by cherry-picking.
 - The “**git dpm**” command provides more enhanced functionality than that of the “**gbp pq**” command.

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

See:

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

Tip



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

7.8.1 gbp import-dscs --debsnap

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

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

7.8.2 Upstream git repository

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

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

Tip



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

Caution

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

7.9 dgit

The new **dgit** package offers commands to automate packaging activities using the git repository as an alternative to still popular **gbp-buildpackage**. Please read their guide:

- **dgit-maint-merge(7)** — for dpkg-source "Format: 3.0 (quilt)" package with its changes flowing both ways between the upstream Git repository and the Debian Git repository which are tightly coupled.
- **dgit-maint-debbase(7)** — for dpkg-source "Format: 3.0 (quilt)" package with its changes flowing mostly one way from the upstream Git repository to the Debian Git repository.
- **dgit-maint-gbp(7)** — for dpkg-source "Format: 3.0 (quilt)" package with its Debian Git repository which is kept usable also for people using **gbp-buildpackage(1)**.
- **dgit-maint-native(7)** — for dpkg-source "Format: 3.0 (native)" package in the Debian Git repository.

The **dgit(1)** command can push the easy-to-trace change history to the <https://browse.dgit.debian.org/> site and can upload Debian package to the Debian repository properly without using **dput(1)**.

- **dgit push-source** — for uploading source-only package (normal)
- **dgit push** — for uploading binary package

7.9.1 Snapshot upstream tarball (alternative git deborig approach)

For **quasi-native** Debian binary package scheme situation described in Section 6.2 can be addressed using alternative **git deborig** approach adopting the **dgit-maint-merge(7)** scheme when **debian/changelog** contains the non-native version number with revision like **0.16-1**.

```
$ cd /path/to/upstream-git
$ git rm -rf debian
$ git tag -s upstream/0.16
$ git reset --hard HEAD^
$ git deborig
$ sbuild
```

Here, for source format **3.0 (quilt)**, removal of files under **debian/** directory in the upstream tarball is non-essential operation to quiet false positive warning from lintian.

For **-1** revision, this use of **git-deborig(1)** as above is how this **debmake-doc** package generates the upstream tarball.

For **-2**, **-3**, ... revisions, you need to fetch and use the uploaded upstream tarball instead. For this, **origtargz(1)** may be handy.

7.10 chroot

The **chroot** for a clean package build environment can be created and managed using the tools described in Chapter 3.¹

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

¹The **sbuild** style chroot organization is deployed here. See <https://wiki.debian.org/sbuild>. Be careful since many HOWTOs use different chroot setups.

- **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
- **sbuid** = **schroot** + **dpkg-buildpackage** (build in the chroot)
- **gbp** = manage the Debian source under git
- **gbp buildpackage** = **sbuid** + **dpkg-buildpackage** + **gbp** (if configured with **sbuid**)

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

- The chroot filesystem creation command for the **unstable** distribution
 - **sudo sbuid-createtchroot --include=auto-apt-proxy unstable /srv/chroot/unstable-amd64-sbuid**
<http://deb.debian.org/debian>
- The master chroot filesystem path for the **unstable** distribution chroot filesystem
 - path: **/srv/chroot/unstable-amd64-sbuid**
- The package build command for the **unstable** distribution chroot
 - **sbuid**
 - **gbp buildpackage**
- The command to update the **unstable** chroot
 - **sudo sbuid-update -udcar u**
- The command to login to the **unstable** chroot filesystem to modify it
 - **sudo sbuid-shell u**

Here, single "u" is predefined short alias for "unstable".

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

- The chroot filesystem creation command for the *dist* distribution
 - **sudo sbuid-createtchroot --include=auto-apt-proxy dist /srv/chroot/dist-amd64-sbuid** <http://deb.debian.org/debian>
- The master chroot filesystem path for the *dist* distribution chroot
 - path: **/srv/chroot/dist-amd64-sbuid**
- The package build command for the *dist* distribution chroot
 - **sbuid -d dist**
 - **gbp buildpackage --git-dist=dist**
- The command to update the *dist* chroot
 - **sudo sbuid-update -udcar dist**
- The command to login to the master chroot (**source:dist**) of the *dist* distribution environment to modify it persistently
 - **sudo sbuid-shell dist**

Tip



For building new experimental packages or for debugging buggy packages without using slow network access, this “**sudo sbuild-shell dist**” command can be used to pre-load some packages. Make sure to retain the backup of the original minimal chroot since changes are persistent and interfere with the normal test package building with the minimal chroot.

Tip



If your old chroot filesystem is missing packages such as **libeatmydata1**, **ccache**, and **lintian**, you may want to install these with the “**sudo sbuild-shell dist**” command.

Tip



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

Note



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

7.11 New Debian revision

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

New Debian revision with the **dquilt** command

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

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

New Debian revision with the **gbp-pq** command

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

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

Tip



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

7.12 New upstream release

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

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

7.12.1 uupdate + tarball

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

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

7.12.2 uscan

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

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

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

7.12.3 gbp

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

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

Tip



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

7.12.4 gbp + uscan

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

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

Tip



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

7.13 3.0 source format

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

- If you need to recreate deleted template files for any reason, you can run **debmake** again in the same Debian package source tree. Then edit them appropriately.
- If the package has not been updated to use the **dh** command for the **debian/rules** file, update it to use it (see Section 5.6.2). Update the **debian/control** file accordingly.
- If you have a **1.0** source package with the **foo.diff.gz** file, you can update it to the newer “**3.0 (quilt)**” source format by creating **debian/source/format** with “**3.0 (quilt)**”. The rest of the **debian/*** files can just be copied. Import the **big.diff** file generated by the “**filterdiff -z -x /debian/ foo.diff.gz > big.diff**” command to your quilt system, if needed.³
- If it was packaged using another patch system such as **dpatch**, **db**s, or **cdb**s with **-p0**, **-p1**, or **-p2**, convert it to the quilt command using the **deb3** script in the **quilt** package.
- If it was packaged with the **dh** command with the “**--with quilt**” option or with the **dh_quilt_patch** and **dh_quilt_unpatch** commands, remove these and make it use the newer “**3.0 (quilt)**” source format.
- If you have a **1.0** source package without the **foo.diff.gz** file, you can update it to the newer “**3.0 (native)**” source format by creating **debian/source/format** with “**3.0 (native)**”. The rest of the **debian/*** files can just be copied.

You should check [DEP - Debian Enhancement Proposals](#) and adopt ACCEPTED proposals.

See [ProjectsDebSrc3.0](#) to check the support status of the new Debian source formats by the Debian tool chains.

7.14 CDBS

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

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

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

Note



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

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

Note



If you are working with a [team](#) of maintainers, please follow the established practice of the team.

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

- [CDBS Documentation](#)
- [The Common Debian Build System \(CDBS\), FOSDEM 2009](#)

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

7.15 Build under UTF-8

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

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

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

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

7.16 UTF-8 conversion

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

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

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

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

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

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

7.17 Upload orig.tar.gz

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

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

- **dpkg-buildpackage -sa**
- **debuild -sa**
- ***sbuild**
- For **gbp buildpackage**, edit the **~/gbp.conf** file.

Tip



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

Tip



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

7.18 Skipped uploads

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

- **dpkg-buildpackage -v1.2**
- **debuild -v1.2**
- **sbuid --debbuildopts -v1.2**
- For **gbp buildpackage**, edit the **~/gbp.conf** file.

7.19 Advanced packaging

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

- differences of the **debhelper** tool behavior under “**compat** <= 8”
- building several binary packages with several different build conditions
 - making multiple copies of the upstream source
 - invoking multiple “**dh_auto_configure -S ...**” commands in the **override_dh_auto_configure** target
 - invoking multiple “**dh_auto_build -S ...**” commands in the **override_dh_auto_build** target
 - invoking multiple “**dh_auto_install -S ...**” commands in the **override_dh_auto_install** target
- building **udeb** packages with “**Package-Type: udeb**” in **debian/control** (see [Package-Type](#))
- excluding some packages for the bootstrapping process (see also [BuildProfileSpec](#))
 - adding the **Build-Profiles** fields in binary package stanzas in **debian/control**
 - building packages with the **DEB_BUILD_PROFILES** environment variable set to the pertinent profile name

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

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

7.20 Other distros

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

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

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

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

If the other distribution of interest is an [RPM](#) based distribution, see [Repackage src.rpm](#).

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

rget script

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

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

7.21 Debug

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

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

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


```
* soft core unlimited
```
 - “**ulimit -c unlimited**” in **~/.bashrc**
 - “**ulimit -a**” to check
 - Press **Ctrl-** or “**kill -ABRT PID**” to make a core dump file
- **gdb** - The GNU Debugger
 - “**info gdb**”
 - “Debugging with GDB” in **/usr/share/doc/gdb-doc/html/gdb/index.html**
- **strace** - Trace system calls and signals
 - Use **strace-graph** script found in **/usr/share/doc/strace/examples/** to make a nice tree view
 - “**man strace**”
- **ltrace** - Trace library calls
 - “**man ltrace**”
- “**sh -n script.sh**” - Syntax check of a Shell script
- “**sh -x script.sh**” - Trace a Shell script
- “**python -m py_compile script.py**” - Syntax check of a Python script
- “**python -mtrace --trace script.py**” - Trace a Python script
- “**perl -I ./libpath -c script.pl**” - Syntax check of a Perl script
- “**perl -d:Trace script.pl**” - Trace a Perl script
 - Install the **libterm-readline-gnu-perl** package or its equivalent to add input line editing capability with history support.
- **lsuf** - List open files by processes
 - “**man lsuf**”

Tip



Der Befehl **script** zeichnet Konsoleausgaben auf.

Tip



Die Befehle **screen** und **tmux** bieten Ihnen im Zusammenspiel mit **ssh** eine sichere und robuste Verbindung zu fernen Terminals.

Tip



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

Tip



Die Befehle **rlwrap** und **rlfe** fügen Eingabebearbeitungsfähigkeiten mit einer Chronik-Unterstützung zu allen interaktiven Befehlen hinzu. Z.B. »**rlwrap dash** -i«.

Chapter 8

Weitere Beispiele

Es gibt ein altes Sprichwort auf Latein: »**fabricando fit faber**« (»Übung macht den Meister«).

Es wird nachdrücklich empfohlen, mit einfachen Paketen zu üben und mit allen Schritten der Debian-Paketierung zu experimentieren. Dieses Kapitel stellt Ihnen viele Fälle von Originalautoren zur Übung bereit.

Es sollte Ihnen auch mit einführenden Beispielen für viele Programmierthemen dienen.

- Programmieren in der POSIX-Shell, Python3 und C
- Methode, um ein Desktop-GUI-Programmstarter mit Icon-Graphiken zu erstellen.
- Umwandlung eines Befehls aus [CLI](#) in [GUI](#).
- Conversion of a program to use **gettext** for [internationalization and localization](#): POSIX shell, Python3, and C sources.
- Überblick über viele Bausysteme: Makefile, Python-Distutils, Autotools und CMake.

Bitte beachten Sie, dass Debian ein paar Dinge sehr Ernst nimmt:

- Freie Software (auch als Libre Software bekannt)
- Stabilität und Sicherheit des Betriebssystems
- Universelles Betriebssystem mittels:
 - freier Wahl der Quellen der Originalautoren
 - freier Wahl der CPU-Architektur und
 - freier Wahl der Sprachen für Oberflächen.

Das in Chapter 4 dargestellte Beispiel ist die Voraussetzung für dieses Kapitel.

Some details are intentionally left vague in the following sections. Please try to read the pertinent documentation and practice yourself to find them out.

Tip



The best source of a packaging example is the current Debian archive itself. Please use the "[Debian Code Search](#)" service to find pertinent examples.

8.1 Cherry-pick templates

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

This is a good platform to get all the template files without making a mess in the upstream source tree you are working on.

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

```
$ mkdir debhello-0.1
$ tree
.
└── debhello-0.1

1 directory, 0 files
```

Let's generate the maximum amount of template files by specifying the **-x4** option.

Let's also use the "**-p debhello -t -u 0.1 -r 1**" options to make the missing upstream tarball.

```
$ debmake -t -p debhello -u 0.1 -r 1 -x4
I: set parameters
I: =====
...
I: sanity check of parameters
W: -p "debhello" != auto set value ""
W: -u "0.1" != auto set value "0~2107020726"
W: -r "1" != auto set value ""
...
I: debmake -x "4" ...
I: creating => debian/control
I: creating => debian/copyright
I: substituting => /usr/share/debmake/extra0/changelog
...
I: creating => debian/license-examples/GPL-3.0+
I: substituting => /usr/share/debmake/extra4/LGPL-2.1+
I: creating => debian/license-examples/LGPL-2.1+
I: substituting => /usr/share/debmake/extra4/LGPL-3.0+
I: creating => debian/license-examples/LGPL-3.0+
I: $ wrap-and-sort
```

Lassen Sie uns die erstellten Vorlagendateien anschauen.

```
$ cd ..
$ tree
.
├── debhello-0.1
└── debian
    ├── README.Debian
    ├── changelog
    ├── clean
    ├── compat.ex
    ├── control
    ├── copyright
    ├── debhello.bug-control.ex
    ├── debhello.bug-presubj.ex
    ├── debhello.bug-script.ex
    ├── debhello.conf-files.ex
    ├── debhello.cron.d.ex
    ├── debhello.cron.daily.ex
    ├── debhello.cron.hourly.ex
    ├── debhello.cron.monthly.ex
    ├── debhello.cron.weekly.ex
    ├── debhello.default.ex
    ├── debhello.dirs.ex
    ├── debhello.emacs-en-install.ex
    ├── debhello.emacs-en-remove.ex
    ├── debhello.emacs-en-startup.ex
    ├── debhello.init.ex
    ├── debhello.lintian-overrides.ex
    ├── debhello.service.ex
    ├── debhello.tmpfile.ex
    ├── debhello.upstart.ex
    └── install
```

```

b''|b''      b''|b''b''-b''b''-b'' license-examples
b''|b''      b''|b''  b''|b''b''-b''b''-b'' Apache-2.0
b''|b''      b''|b''  b''|b''b''-b''b''-b'' Artistic-1.0
b''|b''      b''|b''  b''|b''b''-b''b''-b'' BSD-3-Clause
b''|b''      b''|b''  b''|b''b''-b''b''-b'' Expat
b''|b''      b''|b''  b''|b''b''-b''b''-b'' GPL-2.0+
b''|b''      b''|b''  b''|b''b''-b''b''-b'' GPL-3.0+
b''|b''      b''|b''  b''|b''b''-b''b''-b'' LGPL-2.1+
b''|b''      b''|b''  b''|b''b''-b''b''-b'' LGPL-3.0+
b''|b''      b''|b''b''-b''b''-b'' links
b''|b''      b''|b''b''-b''b''-b'' manpage.1.ex
b''|b''      b''|b''b''-b''b''-b'' manpage.asciidoc.ex
b''|b''      b''|b''b''-b''b''-b'' manpage.sgml.ex
b''|b''      b''|b''b''-b''b''-b'' manpage.xml.ex
b''|b''      b''|b''b''-b''b''-b'' patches
b''|b''      b''|b''  b''|b''b''-b''b''-b'' series
b''|b''      b''|b''b''-b''b''-b'' postinst
b''|b''      b''|b''b''-b''b''-b'' postrm
b''|b''      b''|b''b''-b''b''-b'' preinst
b''|b''      b''|b''b''-b''b''-b'' prerm
b''|b''      b''|b''b''-b''b''-b'' rules
b''|b''      b''|b''b''-b''b''-b'' source
b''|b''      b''|b''  b''|b''b''-b''b''-b'' control
b''|b''      b''|b''  b''|b''b''-b''b''-b'' format
b''|b''      b''|b''  b''|b''b''-b''b''-b'' local-options
b''|b''      b''|b''  b''|b''b''-b''b''-b'' options
b''|b''      b''|b''  b''|b''b''-b''b''-b'' patch-header
b''|b''      b''|b''b''-b''b''-b'' source.lintian-overrides.ex
b''|b''      b''|b''b''-b''b''-b'' tests
b''|b''      b''|b''  b''|b''b''-b''b''-b'' control
b''|b''      b''|b''b''-b''b''-b'' upstream
b''|b''      b''|b''  b''|b''b''-b''b''-b'' metadata
b''|b''      b''|b''b''-b''b''-b'' watch
b''|b''b''-b''b''-b'' debhello-0.1.tar.gz
b''|b''b''-b''b''-b'' debhello_0.1.orig.tar.gz -> debhello-0.1.tar.gz

7 directories, 56 files

```

Now you can copy any of these generated template files in the *debhello-0.1/debian/* directory to your package as needed while renaming them as needed.

Tip



The generated template files can be made more verbose by invoking the **debmake** command with the **-T** option (tutorial mode).

8.2 Kein Makefile (Shell, CLI)

Here is an example of creating a simple Debian package from a POSIX shell CLI program without its build system.

Let's assume this upstream tarball to be **debhello-0.2.tar.gz**.

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

For example:

```

$ tar -xzmf debhello-0.2.tar.gz
$ cd 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.

Laden Sie debhello-0.2.tar.gz herunter

```
$ wget http://www.example.org/download/debhello-0.2.tar.gz
...
$ tar -xzmf debhello-0.2.tar.gz
$ tree
.
├── debhello-0.2
│   ├── LICENSE
│   ├── data
│   ├── hello.desktop
│   ├── hello.png
│   ├── man
│   ├── hello.1
│   ├── scripts
│   └── hello
└── debhello-0.2.tar.gz

4 directories, 6 files
```

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

hello (v=0.2)

```
$ cat debhello-0.2/scripts/hello
#!/bin/sh -e
echo "Hello from the shell!"
echo ""
echo -n "Type Enter to exit this program: "
read X
```

Here, **hello.desktop** supports the [Desktop Entry Specification](#).

hello.desktop (v=0.2)

```
$ cat debhello-0.2/data/hello.desktop
[Desktop Entry]
Name=Hello
Name[fr]=Bonjour
Comment=Greetings
Comment[fr]=Salutations
Type=Application
Keywords=hello
Exec=hello
Terminal=true
Icon=hello.png
Categories=Utility;
```

Here, **hello.png** is the icon graphics file.

Let's package this with the **debmake** command. Here, the **-b':sh'** option is used to specify that the generated binary package is a shell script.

```
$ cd debhello-0.2
$ debmake -b':sh'
I: set parameters
...
I: sanity check of parameters
I: pkg="debhello", ver="0.2", rev="1"
I: *** start packaging in "debhello-0.2". ***
I: provide debhello_0.2.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-0.2.tar.gz debhello_0.2.orig.tar.gz
I: pwd = "/path/to/debhello-0.2"
I: parse binary package settings: :sh
I: binary package=debhello Type=script / Arch=all M-A=foreign
I: analyze the source tree
I: build_type = Unknown
I: scan source for copyright+license text and file extensions
I: 33 %, ext = desktop
```

...

Let's inspect notable template files generated.

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

```
$ cd ..
$ tree
.
├── debhello-0.2
│   ├── LICENSE
│   ├── data
│   ├── hello.desktop
│   ├── hello.png
│   ├── debian
│   │   ├── README.Debian
│   │   ├── changelog
│   │   ├── control
│   │   ├── copyright
│   │   ├── patches
│   │   ├── series
│   │   ├── rules
│   │   └── source
│   ├── control
│   ├── format
│   ├── local-options
│   ├── options
│   ├── patch-header
│   ├── tests
│   ├── control
│   ├── upstream
│   ├── metadata
│   ├── watch
│   ├── man
│   ├── hello.1
│   ├── scripts
│   └── hello
└── debhello-0.2.tar.gz
    └── debhello_0.2.orig.tar.gz -> debhello-0.2.tar.gz

9 directories, 21 files
```

debian/rules (Vorlagendatei, v=0.2):

```
$ cat debhello-0.2/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1

%:
    dh $@
```

This is essentially the standard **debian/rules** file with the **dh** command. Since this is the script package, this template **debian/rules** file has no build flag related contents.

debian/control (Vorlagendatei, v=0.2):

```
$ cat debhello-0.2/debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: "Firstname Lastname" <email.address@example.org>
Build-Depends: debhelper-compat (= 13)
Standards-Version: 4.5.1
Homepage: <insert the upstream URL, if relevant>
Rules-Requires-Root: no
```

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

Since this is the shell script package, the **debmake** command sets “**Architecture: all**” and “**Multi-Arch: foreign**”. Also, it sets required **substvar** parameters as “**Depends: \${misc:Depends}**”. These are explained in Chapter 5.

Since this upstream source lacks the upstream **Makefile**, that functionality needs to be provided by the maintainer. This upstream source contains only a script file and data files and no C source files; the **build** process can be skipped but the **install** process needs to be implemented. For this case, this is achieved cleanly by adding the **debian/install** and **debian/manpages** files without complicating the **debian/rules** file.

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

debian/rules (Betreuerversion, v=0.2):

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

%:
    dh $@
```

debian/control (Betreuerversion, v=0.2):

```
$ vim debhello-0.2/debian/control
... hack, hack, hack, ...
$ cat debhello-0.2/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper-compat (= 13)
Standards-Version: 4.5.1
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)
```

Warning



If you leave “**Section: unknown**” in the template **debian/control** file unchanged, the **lintian** error may cause a build failure.

debian/install (Betreuerversion, v=0.2):

```
$ vim debhello-0.2/debian/install
... hack, hack, hack, ...
$ cat debhello-0.2/debian/install
data/hello.desktop usr/share/applications
```

```
data/hello.png usr/share/pixmaps
scripts/hello usr/bin
```

debian/manpages (Betreuerversion, v=0.2):

```
$ vim debhello-0.2/debian/manpages
... hack, hack, hack, ...
$ cat debhello-0.2/debian/manpages
man/hello.1
```

Es gibt eine Reihe von weiteren Vorlagendateien unter dem Verzeichnis **debian/**. Diese müssen auch aktualisiert werden.

Vorlagendateien unter debian/. (v=0.2):

```
$ tree debhello-0.2/debian
debhello-0.2/debian
b' |-b' |-b' |-b' |-b' README.Debian
b' |-b' |-b' |-b' |-b' changelog
b' |-b' |-b' |-b' |-b' control
b' |-b' |-b' |-b' |-b' copyright
b' |-b' |-b' |-b' |-b' install
b' |-b' |-b' |-b' |-b' manpages
b' |-b' |-b' |-b' |-b' patches
b' |-b' |-b' |-b' |-b' series
b' |-b' |-b' |-b' |-b' rules
b' |-b' |-b' |-b' |-b' source
b' |-b' |-b' |-b' |-b' control
b' |-b' |-b' |-b' |-b' format
b' |-b' |-b' |-b' |-b' local-options
b' |-b' |-b' |-b' |-b' options
b' |-b' |-b' |-b' |-b' patch-header
b' |-b' |-b' |-b' |-b' tests
b' |-b' |-b' |-b' |-b' control
b' |-b' |-b' |-b' |-b' upstream
b' |-b' |-b' |-b' |-b' metadata
b' |-b' |-b' |-b' |-b' watch

4 directories, 16 files
```

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

```
$ cd debhello-0.2
$ debuild
dpkg-buildpackage -us -uc -ui -i -i
...
debian/rules clean
dh clean
...
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_install
install -d debian/debhello/usr/bin
cp --reflink=auto -a ./scripts/hello debian/debhello/usr/bin/
...
```

Lassen Sie uns die Ergebnisse anschauen.

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

```

$ cd ..
$ tree -FL 1
.
|b'|b'|b'|b'|b'|b'|b'| debhello-0.2/
|b'|b'|b'|b'|b'|b'|b'| debhello-0.2.tar.gz
|b'|b'|b'|b'|b'|b'|b'| debhello_0.2-1.debian.tar.xz
|b'|b'|b'|b'|b'|b'|b'| debhello_0.2-1.dsc
|b'|b'|b'|b'|b'|b'|b'| debhello_0.2-1_all.deb
|b'|b'|b'|b'|b'|b'|b'| debhello_0.2-1_amd64.build
|b'|b'|b'|b'|b'|b'|b'| debhello_0.2-1_amd64.buildinfo
|b'|b'|b'|b'|b'|b'|b'| debhello_0.2-1_amd64.changes
|b'|b'|b'|b'|b'|b'|b'| debhello_0.2.orig.tar.gz -> debhello-0.2.tar.gz
1 directory, 8 files

```

You see all the generated files.

- The **debhello_0.2.orig.tar.gz** file is a symlink to the upstream tarball.
- The **debhello_0.2-1.debian.tar.xz** file contains the maintainer generated contents.
- The **debhello_0.2-1.dsc** file is the meta data file for the Debian source package.
- Die Datei **debhello_0.2-1_all.deb** ist das Debian-Binärpaket.
- The **debhello_0.2-1_amd64.build** file is the build log file.
- The **debhello_0.2-1_amd64.buildinfo** file is the meta data file generated by **dpkg-genbuildinfo(1)**.
- The **debhello_0.2-1_amd64.changes** file is the meta data file for the Debian binary package.

The **debhello_0.2-1.debian.tar.xz** file contains the Debian changes to the upstream source as follows.

The compressed archive contents of debhello_0.2-1.debian.tar.xz:

```

$ tar -tzf debhello-0.2.tar.gz
debhello-0.2/
debhello-0.2/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/LICENSE
$ tar --xz -tf debhello_0.2-1.debian.tar.xz
debian/
debian/README.Debian
debian/changelog
debian/control
debian/copyright
debian/install
debian/manpages
debian/patches/
debian/patches/series
debian/rules
debian/source/
debian/source/control
debian/source/format
debian/source/options
debian/source/patch-header
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:

```
$ 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
```

Dies ist die erstellte Abhängigkeitsliste von **debhello_0.2-1_all.deb**.

The generated dependency list of debhello_0.2-1_all.deb:

```
$ dpkg -f debhello_0.2-1_all.deb pre-depends depends recommends conflicts br...
```

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

Note



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)`.

8.3 Makefile (Shell, CLI)

Here is an example of creating a simple Debian package from a POSIX shell CLI program using the **Makefile** as its build system.

Let's assume its upstream tarball to be **debhello-1.0.tar.gz**.

Diese Art der Quellen soll als Nichtsystemdatei wie folgt installiert werden:

```
$ tar -xzmf debhello-1.0.tar.gz
$ cd debhello-1.0
$ make install
```

Die Debian-Paketierung ändert diesen »**make install**«-Prozess in die Installation von Dateien an den Ort im Zielsystemabbild statt des normalen Ortes unterhalb von **/usr/local**.

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

Laden Sie debhello-1.0.tar.gz herunter

```
$ wget http://www.example.org/download/debhello-1.0.tar.gz
...
$ tar -xzmf debhello-1.0.tar.gz
$ tree
.
b'|-b'b'|-b'b'|-b' debhello-1.0
b'|b' b'|-b'b'|-b'b'|-b' LICENSE
b'|b' b'|-b'b'|-b'b'|-b' Makefile
b'|b' b'|-b'b'|-b'b'|-b' data
b'|b' b'|b' b'|-b'b'|-b'b'|-b' hello.desktop
```

```

b''|b''  b''|b''  b''|b''b''-b''b''-b'' hello.png
b''|b''  b''|b''b''-b''b''-b'' man
b''|b''  b''|b''  b''|b''b''-b''b''-b'' hello.1
b''|b''  b''|b''b''-b''b''-b'' scripts
b''|b''  b''|b''b''-b''b''-b'' hello
b''|b''b''-b''b''-b'' debhello-1.0.tar.gz

```

4 directories, 7 files

Here, the **Makefile** uses **\$(DESTDIR)** and **\$(prefix)** properly. All other files are the same as in Section 8.2 and most of the packaging activities are the same.

Makefile (v=1.0)

```

$ cat debhello-1.0/Makefile
prefix = /usr/local

all:
    : # do nothing

install:
    install -D scripts/hello \
        $(DESTDIR)$(prefix)/bin/hello
    install -m 644 -D data/hello.desktop \
        $(DESTDIR)$(prefix)/share/applications/hello.desktop
    install -m 644 -D data/hello.png \
        $(DESTDIR)$(prefix)/share/pixmaps/hello.png
    install -m 644 -D man/hello.1 \
        $(DESTDIR)$(prefix)/share/man/man1/hello.1

clean:
    : # do nothing

distclean: clean

uninstall:
    -rm -f $(DESTDIR)$(prefix)/bin/hello
    -rm -f $(DESTDIR)$(prefix)/share/applications/hello.desktop
    -rm -f $(DESTDIR)$(prefix)/share/pixmaps/hello.png
    -rm -f $(DESTDIR)$(prefix)/share/man/man1/hello.1

.PHONY: all install clean distclean uninstall

```

Let's package this with the **debmake** command. Here, the **-b':sh'** option is used to specify that the generated binary package is a shell script.

```

$ cd debhello-1.0
$ debmake -b':sh' -e osamu@debian.org -f 'Osamu Aoki'
I: set parameters
...
I: sanity check of parameters
I: pkg="debhello", ver="1.0", rev="1"
I: *** start packaging in "debhello-1.0". ***
I: provide debhello_1.0.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.0.tar.gz debhello_1.0.orig.tar.gz
I: pwd = "/path/to/debhello-1.0"
I: parse binary package settings: :sh
I: binary package=debhello Type=script / Arch=all M-A=foreign
I: analyze the source tree
I: build_type = make
I: scan source for copyright+license text and file extensions
I: 33 %, ext = desktop
...

```

Let's inspect the notable template files generated.

debian/rules (Vorlagendatei, v=1.0):

```
$ cat debhello-1.0/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1

%:
    dh $@

#override_dh_auto_install:
#    dh_auto_install -- prefix=/usr

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

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

debian/rules (Betreuerversion, v=1.0):

```
$ vim debhello-1.0/debian/rules
... hack, hack, hack, ...
$ cat debhello-1.0/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1

%:
    dh $@

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

Since this upstream source has the proper upstream **Makefile**, there is no need to create **debian/install** and **debian/manpages** files.

The **debian/control** file is exactly the same as the one in Section 8.2.

Es gibt eine Reihe von weiteren Vorlagendateien unter dem Verzeichnis **debian/**. Diese müssen auch aktualisiert werden.

Vorlagendateien unter debian/. (v=1.0):

```
$ tree debhello-1.0/debian
debhello-1.0/debian
├── README.Debian
├── changelog
├── control
├── copyright
├── patches
├── series
├── rules
├── source
├── control
├── format
├── local-options
├── options
├── patch-header
├── tests
├── control
├── upstream
├── metadata
└── watch

4 directories, 14 files
```

The rest of the packaging activities are practically the same as the ones in Section 8.2.


```
$ cat debhello-1.1/setup.cfg
[metadata]
name = debhello
version = attr: debhello.__version__
description = Hello Python (CLI)
long_description = file: README.md
classifiers =
    Development Status :: 5 - Production/Stable
    Intended Audience :: Developers
    Topic :: System :: Archiving :: Packaging
    License :: OSI Approved :: MIT License
    Programming Language :: Python
    Programming Language :: Python :: 3
    Programming Language :: Python :: 3.5
    Programming Language :: Python :: 3.6
    Programming Language :: Python :: 3.7
    Programming Language :: Python :: 3.8
    Programming Language :: Python :: 3.9
author = Osamu Aoki
author_email = osamu@debian.org
maintainer = Osamu Aoki
maintainer_email = osamu@debian.org
url = https://salsa.debian.org/debian/debmake-doc
project_urls =
    Documentation = https://salsa.debian.org/debian/debmake-doc
keywords = debhello
license = MIT

[options]
package_dir =
    = src
packages = find:
python_requires = >=3.4
setup_requires = setuptools >= 42

[options.packages.find]
where = src

[options.entry_points]
console_scripts =
    hello = debhello.__main__:main
```

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

```
$ cat debhello-1.1/pyproject.toml
[build-system]
requires = [
    "setuptools>=42",
    "wheel"
]
build-backend = "setuptools.build_meta"
```

MANIFEST.in (v=1.1) — for distutils to make sdist

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

src/debhello/__init__.py (v=1.1) — program version definition

```
$ cat debhello-1.1/src/debhello/__init__.py
__version__ = '1.1.0'
```

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

```
$ cat debhello-1.1/src/debhello/__main__.py
"""
debhello command line entry (enable python -m debhello syntax)
"""

import sys
import debhello

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.

```
$ cd debhello-1.1
$ debmake -b':py3' -e osamu@debian.org -f 'Osamu Aoki'
I: set parameters
...
I: sanity check of parameters
I: pkg="debhello", ver="1.1", rev="1"
I: *** start packaging in "debhello-1.1". ***
I: provide debhello_1.1.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.1.tar.gz debhello_1.1.orig.tar.gz
I: pwd = "/path/to/debhello-1.1"
I: parse binary package settings: :py3
I: binary package=debhello Type=python3 / Arch=all M-A=foreign
I: analyze the source tree
W: unknown python version. check setup.py.
I: build_type =
I: scan source for copyright+license text and file extensions
...
```

Let's inspect the notable template files generated.

debian/rules (Vorlagendatei, v=1.1):

```
$ cat debhello-1.1/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1

%:
    dh $@ --with python3
```

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 (Vorlagendatei, v=1.1):

```
$ cat debhello-1.1/debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper-compat (= 13), dh-python, python3-all
```

```
Standards-Version: 4.5.1
Homepage: <insert the upstream URL, if relevant>
Rules-Requires-Root: no

Package: debhello
Architecture: all
Multi-Arch: foreign
Depends: ${misc:Depends}, ${python3:Depends}
Description: auto-generated package by debmake
 This Debian binary package was auto-generated by the
 debmake(1) command provided by the debmake package.
```

Since this is the Python3 package, the **debmake** command sets “**Architecture: all**” and “**Multi-Arch: foreign**”. Also, it sets required **substvar** parameters as “**Depends: \${python3:Depends}, \${misc:Depends}**”. These are explained in Chapter 5.

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

debian/rules (Betreuerversion, v=1.1):

```
$ vim debhello-1.1/debian/rules
... hack, hack, hack, ...
$ cat debhello-1.1/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 (Betreuerversion, v=1.1):

```
$ vim debhello-1.1/debian/control
... hack, hack, hack, ...
$ cat debhello-1.1/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper-compat (= 13),
               dh-python,
               python3-all,
               python3-setuptools
Standards-Version: 4.5.1
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)'.
```

Es gibt eine Reihe von weiteren Vorlagendateien unter dem Verzeichnis **debian/**. Diese müssen auch aktualisiert werden.

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

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

```
$ vim debhello-1.1/debian/install
```

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

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

```
$ vim debhello-1.1/debian/manpages
... hack, hack, hack, ...
$ cat debhello-1.1/debian/manpages
manpages/hello.1
```

The rest of the packaging activities are practically the same as the ones in Section 8.3.

Vorlagendateien unter debian/. (v=1.1):

```
$ tree debhello-1.1/debian
debhello-1.1/debian
b'''|b'''b'''-b'''b'''-b''' README.Debian
b'''|b'''b'''-b'''b'''-b''' changelog
b'''|b'''b'''-b'''b'''-b''' control
b'''|b'''b'''-b'''b'''-b''' copyright
b'''|b'''b'''-b'''b'''-b''' install
b'''|b'''b'''-b'''b'''-b''' manpages
b'''|b'''b'''-b'''b'''-b''' patches
b'''|b'''b'''-b'''b'''-b'''b'''-b''' series
b'''|b'''b'''-b'''b'''-b''' rules
b'''|b'''b'''-b'''b'''-b''' source
b'''|b'''b'''b'''|b'''b'''b'''-b'''b'''-b''' control
b'''|b'''b'''b'''|b'''b'''b'''-b'''b'''-b''' format
b'''|b'''b'''b'''|b'''b'''b'''-b'''b'''-b''' local-options
b'''|b'''b'''b'''|b'''b'''b'''-b'''b'''-b''' options
b'''|b'''b'''b'''|b'''b'''b'''-b'''b'''-b''' patch-header
b'''|b'''b'''-b'''b'''-b''' tests
b'''|b'''b'''b'''|b'''b'''b'''-b'''b'''-b''' control
b'''|b'''b'''-b'''b'''-b''' upstream
b'''|b'''b'''b'''|b'''b'''b'''-b'''b'''-b''' metadata
b'''|b'''b'''-b'''b'''-b''' watch

4 directories, 16 files
```

Hier ist die erstellte Abhängigkeitsliste von **debhello 1.1-1 all.deb**.

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

```
$ dpkg -f debhello_1.1-1_all.deb pre-depends depends recommends conflicts br...
Depends: python3:any
```

8.5 Makefile (Shell, GUI)

Here is an example of creating a simple Debian package from a POSIX shell GUI program using the **Makefile** as its build system.

This upstream is based on Section 8.3 with enhanced GUI support.

Let's assume its upstream tarball to be **debhello-1.2.tar.gz**.

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

Laden Sie debhello-1.2.tar.gz herunter

```
$ wget http://www.example.org/download/debhello-1.2.tar.gz
...
$ tar -xzmf debhello-1.2.tar.gz
$ tree
.
├── 'b''b''-b''b''-b'' debhello-1.2
├── 'b''      b''|b''b''-b''b''-b'' LICENSE
└── 'b''      b''|b''b''-b''b''-b'' Makefile
```



```

b''|b''  b''|b''b''-b''b''-b'' data
b''|b''  b''|b''  b''|b''b''-b''b''-b'' hello.desktop
b''|b''  b''|b''  b''|b''b''-b''b''-b'' hello.png
b''|b''  b''|b''b''-b''b''-b'' man
b''|b''  b''|b''  b''|b''b''-b''b''-b'' hello.1
b''|b''  b''|b''b''-b''b''-b'' scripts
b''|b''  b''|b''b''-b''b''-b'' hello
b''|b''b''-b''b''-b'' debhello-1.2.tar.gz

```

4 directories, 7 files

Here, the **hello** has been re-written to use the **zenity** command to make this a GTK+ GUI program.

hello (v=1.2)

```

$ cat debhello-1.2/scripts/hello
#!/bin/sh -e
zenity --info --title "hello" --text "Hello from the shell!"

```

Here, the desktop file is updated to be **Terminal=false** as a GUI program.

hello.desktop (v=1.2)

```

$ cat debhello-1.2/data/hello.desktop
[Desktop Entry]
Name=Hello
Name[fr]=Bonjour
Comment=Greetings
Comment[fr]=Salutations
Type=Application
Keywords=hello
Exec=hello
Terminal=false
Icon=hello.png
Categories=Utility;

```

Alle anderen Dateien sind identisch zu Section 8.3.

Let's package this with the **debmake** command. Here, the **-b':sh'** option is used to specify that the generated binary package is a shell script.

```

$ cd debhello-1.2
$ debmake -b':sh' -e osamu@debian.org -f 'Osamu Aoki'
I: set parameters
...
I: sanity check of parameters
I: pkg="debhello", ver="1.2", rev="1"
I: *** start packaging in "debhello-1.2". ***
I: provide debhello_1.2.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.2.tar.gz debhello_1.2.orig.tar.gz
I: pwd = "/path/to/debhello-1.2"
I: parse binary package settings: :sh
I: binary package=debhello Type=script / Arch=all M-A=foreign
I: analyze the source tree
I: build_type = make
I: scan source for copyright+license text and file extensions
I: 33 %, ext = desktop
...

```

Let's inspect the notable template files generated.

debian/control (Vorlagendatei, v=1.2):

```

$ cat debhello-1.2/debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper-compat (= 13)

```

```
Standards-Version: 4.5.1
Homepage: <insert the upstream URL, if relevant>
Rules-Requires-Root: no

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

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

debian/control (Betreuerversion, v=1.2):

```
$ vim debhello-1.2/debian/control
... hack, hack, hack, ...
$ cat debhello-1.2/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper-compat (= 13)
Standards-Version: 4.5.1
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 Section 8.3.

Es gibt eine Reihe von weiteren Vorlagendateien unter dem Verzeichnis **debian/**. Diese müssen auch aktualisiert werden.

Template files under debian/. (v=1.2):

```
$ tree debhello-1.2/debian
debhello-1.2/debian
├── README.Debian
├── changelog
├── control
├── copyright
├── patches
├── series
├── rules
├── source
├── control
│   ├── control
│   ├── format
│   ├── local-options
│   ├── options
│   └── patch-header
├── tests
├── control
│   └── upstream
├── metadata
└── watch

4 directories, 14 files
```

Die restlichen Paketieraktivitäten sind praktisch die gleichen wie in Section 8.3.

Hier ist die erstellte Abhängigkeitsliste von **debhello_1.2-1_all.deb**.

The generated dependency list of debhello_1.2-1_all.deb:

```
$ dpkg -f debhello_1.2-1_all.deb pre-depends depends recommends conflicts br...
Depends: zenity
```

8.6 setup.py (Python3, GUI)

Here is an example of creating a simple Debian package from a Python3 GUI program using the **setuptools** based **setup.py** as its build system.

This upstream is based on Section 8.4 with enhanced GUI, desktop icon, and manpage support.

Let's assume this upstream tarball to be **debhello-1.3.tar.gz**.

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

Download debhello-1.3.tar.gz

```
$ wget http://www.example.org/download/debhello-1.3.tar.gz
...
$ tar -xzmf debhello-1.3.tar.gz
$ tree
.
b' | b' b' '-b' b' '-b' debhello-1.3
b' | b' b' b' | b' b' '-b' b' '-b' LICENSE
b' | b' b' b' | b' b' '-b' b' '-b' MANIFEST.in
b' | b' b' b' | b' b' '-b' b' '-b' README.md
b' | b' b' b' | b' b' '-b' b' '-b' data
b' | b' b' b' | b' b' b' | b' b' '-b' b' '-b' hello.desktop
b' | b' b' b' | b' b' b' | b' b' '-b' b' '-b' hello.png
b' | b' b' b' | b' b' '-b' b' '-b' manpages
b' | b' b' b' | b' b' b' | b' b' '-b' b' '-b' hello.1
b' | b' b' b' | b' b' '-b' b' '-b' pyproject.toml
b' | b' b' b' | b' b' '-b' b' '-b' setup.cfg
b' | b' b' b' | b' b' '-b' b' '-b' setup.py
b' | b' b' b' | b' b' '-b' b' '-b' src
b' | b' b' b' | b' b' '-b' b' '-b' debhello
b' | b' b' b' | b' b' b' | b' b' '-b' b' '-b' __init__.py
b' | b' b' b' | b' b' b' | b' b' '-b' b' '-b' __main__.py
b' | b' b' b' '-b' b' '-b' debhello-1.3.tar.gz

5 directories, 12 files
```

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

setup.py (v=1.3) — a boiler plate setup.py for the backward compatibility

```
$ cat debhello-1.3/setup.py
import setuptools
setuptools.setup()
```

setup.cfg (v=1.3) — modern static metadata

```
$ cat debhello-1.3/setup.cfg
[metadata]
name = debhello
version = attr: debhello.__version__
description = Hello Python (GUI)
long_description = file: README.md
classifiers =
    Development Status :: 5 - Production/Stable
    Intended Audience :: Developers
    Topic :: System :: Archiving :: Packaging
    License :: OSI Approved :: MIT License
    Programming Language :: Python
    Programming Language :: Python :: 3
    Programming Language :: Python :: 3.5
```

```

        Programming Language :: Python :: 3.6
        Programming Language :: Python :: 3.7
        Programming Language :: Python :: 3.8
        Programming Language :: Python :: 3.9
author = Osamu Aoki
author_email = osamu@debian.org
maintainer = Osamu Aoki
maintainer_email = osamu@debian.org
url = https://salsa.debian.org/debian/debmake-doc
project_urls =
    Documentation = https://salsa.debian.org/debian/debmake-doc
keywords = debhello
license = MIT

[options]
package_dir =
    = src
packages = find:
python_requires = >=3.4
setup_requires = setuptools >= 42

[options.packages.find]
where = src

[options.entry_points]
console_scripts =
    hello = debhello.__main__:main

```

pyproject.toml (v=1.3) — PEP 517 configuration

```

$ cat debhello-1.3/pyproject.toml
[build-system]
requires = [
    "setuptools>=42",
    "wheel"
]
build-backend = "setuptools.build_meta"

```

MANIFEST.in (v=1.3) — for distutils to make sdist

```

$ cat debhello-1.3/MANIFEST.in
include data/*
include manpages/*

```

src/debhello/__init__.py (v=1.3) — program version definition

```

$ cat debhello-1.3/src/debhello/__init__.py
__version__ = '1.3.0'

```

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

```

$ cat debhello-1.3/src/debhello/__main__.py
#!/usr/bin/python3
from gi.repository import Gtk

class TopWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self)
        self.title = "Hello World!"
        self.counter = 0
        self.border_width = 10
        self.set_default_size(400, 100)
        self.set_position(Gtk.WindowPosition.CENTER)
        self.button = Gtk.Button(label="Click me!")
        self.button.connect("clicked", self.on_button_clicked)

```

```

        self.add(self.button)
        self.connect("delete-event", self.on_window_destroy)

    def on_window_destroy(self, *args):
        Gtk.main_quit(*args)

    def on_button_clicked(self, widget):
        self.counter += 1
        widget.set_label("Hello, World!\nClick count = %i" % self.counter)

def main():
    window = TopWindow()
    window.show_all()
    Gtk.main()

if __name__ == '__main__':
    main()

```

Let's package this with the **debmake** command. Here, the **-b':py3'** option is used to specify that the generated binary package contains Python3 script and module files.

```

$ cd debhello-1.3
$ debmake -b':py3' -e osamu@debian.org -f 'Osamu Aoki'
I: set parameters
...
I: sanity check of parameters
I: pkg="debhello", ver="1.3", rev="1"
I: *** start packaging in "debhello-1.3". ***
I: provide debhello_1.3.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.3.tar.gz debhello_1.3.orig.tar.gz
I: pwd = "/path/to/debhello-1.3"
I: parse binary package settings: :py3
I: binary package=debhello Type=python3 / Arch=all M-A=foreign
I: analyze the source tree
W: unknown python version. check setup.py.
I: build_type =
I: scan source for copyright+license text and file extensions
...

```

The result is practically the same as in Section 8.4.

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

debian/rules (Betreuerversion, v=1.3):

```

$ vim debhello-1.3/debian/rules
... hack, hack, hack, ...
$ cat debhello-1.3/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 (Betreuerversion, v=1.3):

```

$ vim debhello-1.3/debian/control
... hack, hack, hack, ...
$ cat debhello-1.3/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper-compat (= 13), dh-python, python3-all

```

```
Standards-Version: 4.5.1
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.

Die restlichen Paketieraktivitäten sind praktisch die gleichen wie in Section 8.4.

Hier ist die erstellte Abhängigkeitsliste von **debhello_1.3-1_all.deb**.

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

```
$ dpkg -f debhello_1.3-1_all.deb pre-depends depends recommends conflicts br...
Depends: gir1.2-gtk-3.0, python3-gi, python3:any
```

8.7 Makefile (Paket mit einem Programm)

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

This is an enhanced upstream source example for Chapter 4. This comes with the manpage, the desktop file, and the desktop icon. This also links to an external library **libm** to be a more practical example.

Let's assume this upstream tarball to be **debhello-1.4.tar.gz**.

Diese Art der Quellen soll als Nichtsystemdatei wie folgt installiert werden:

```
$ tar -xzf debhello-1.4.tar.gz
$ cd debhello-1.4
$ make
$ make install
```

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

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

Laden Sie debhello-1.4.tar.gz herunter

```
$ wget http://www.example.org/download/debhello-1.4.tar.gz
...
$ tar -xzf debhello-1.4.tar.gz
$ tree
.
|__|b''b''-b''b''-b'' debhello-1.4
|__|b'' b''|b''b''-b''b''-b'' LICENSE
|__|b'' b''|b''b''-b''b''-b'' Makefile
|__|b'' b''|b''b''-b''b''-b'' data
|__|b'' b''|b'' b''|b''b''-b''b''-b'' hello.desktop
|__|b'' b''|b''|b'' b''|b''b''-b''b''-b'' hello.png
|__|b'' b''|b''|b''b''-b''b''-b'' man
|__|b'' b''|b''|b'' b''|b''b''-b''b''-b'' hello.1
|__|b'' b''|b''|b''b''-b''b''-b'' src
|__|b'' b''|b''|b''b''-b''b''-b'' config.h
|__|b'' b''|b''|b''b''-b''b''-b'' hello.c
|__|b''b''b''-b''b''-b'' debhello-1.4.tar.gz

4 directories, 8 files
```

Here, the contents of this source are as follows.

src/hello.c (v=1.4):

```
$ cat debhello-1.4/src/hello.c
#include "config.h"
#include <math.h>
#include <stdio.h>
int
main()
{
    printf("Hello, I am " PACKAGE_AUTHOR "!\n");
    printf("4.0 * atan(1.0) = %10f8\n", 4.0*atan(1.0));
    return 0;
}
```

src/config.h (v=1.4):

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

Makefile (v=1.4):

```
$ cat debhello-1.4/Makefile
prefix = /usr/local

all: src/hello

src/hello: src/hello.c
    $(CC) $(CPPFLAGS) $(CFLAGS) $(LDFLAGS) -o $@ $^ -lm

install: src/hello
    install -D src/hello \
        $(DESTDIR)$(prefix)/bin/hello
    install -m 644 -D data/hello.desktop \
        $(DESTDIR)$(prefix)/share/applications/hello.desktop
    install -m 644 -D data/hello.png \
        $(DESTDIR)$(prefix)/share/pixmaps/hello.png
    install -m 644 -D man/hello.1 \
        $(DESTDIR)$(prefix)/share/man/man1/hello.1

clean:
    -rm -f src/hello

distclean: clean

uninstall:
    -rm -f $(DESTDIR)$(prefix)/bin/hello
    -rm -f $(DESTDIR)$(prefix)/share/applications/hello.desktop
    -rm -f $(DESTDIR)$(prefix)/share/pixmaps/hello.png
    -rm -f $(DESTDIR)$(prefix)/share/man/man1/hello.1

.PHONY: all install clean distclean uninstall
```

Please note that this **Makefile** has the proper **install** target for the manpage, the desktop file, and the desktop icon.

Let's package this with the **debmake** command.

```
$ cd debhello-1.4
$ debmake -e osamu@debian.org -f 'Osamu Aoki'
I: set parameters
...
I: sanity check of parameters
I: pkg="debhello", ver="1.4", rev="1"
I: *** start packaging in "debhello-1.4". ***
I: provide debhello_1.4.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.4.tar.gz debhello_1.4.orig.tar.gz
I: pwd = "/path/to/debhello-1.4"
```

```
I: parse binary package settings:
I: binary package=debhello Type=bin / Arch=any M-A=foreign
I: analyze the source tree
I: build_type = make
I: scan source for copyright+license text and file extensions
I: 40 %, ext = c
...
```

The result is practically the same as in Section 4.6.

Let's make this Debian package, which is practically the same as in Section 4.7, 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 Section 4.7, since the **libm** library is always available as a part of **libc6** (Priority: required).

Es gibt eine Reihe von weiteren Vorlagendateien unter dem Verzeichnis **debian/**. Diese müssen auch aktualisiert werden.

Vorlagendateien unter debian/. (v=1.4):

```
$ tree debhello-1.4/debian
debhello-1.4/debian
b' |-b' |-b' |-b' |-b' README.Debian
b' |-b' |-b' |-b' |-b' changelog
b' |-b' |-b' |-b' |-b' control
b' |-b' |-b' |-b' |-b' copyright
b' |-b' |-b' |-b' |-b' patches
b' |-b' |-b' |-b' |-b' series
b' |-b' |-b' |-b' |-b' rules
b' |-b' |-b' |-b' |-b' source
b' |-b' |-b' |-b' |-b' control
b' |-b' |-b' |-b' |-b' format
b' |-b' |-b' |-b' |-b' local-options
b' |-b' |-b' |-b' |-b' options
b' |-b' |-b' |-b' |-b' patch-header
b' |-b' |-b' |-b' |-b' tests
b' |-b' |-b' |-b' |-b' control
b' |-b' |-b' |-b' |-b' upstream
b' |-b' |-b' |-b' |-b' metadata
b' |-b' |-b' |-b' |-b' watch

4 directories, 14 files
```

The rest of the packaging activities are practically the same as the one in Section 4.8.

Here is the generated dependency list of all binary packages.

Die erstellte Abhängigkeitsliste aller Binärpakete (v=1.4):

```
$ dpkg -f debhello-dbgsym_1.4-1_amd64.deb pre-depends depends recommends con...
Depends: debhello (= 1.4-1)
$ dpkg -f debhello_1.4-1_amd64.deb pre-depends depends recommends conflicts ...
Depends: libc6 (>= 2.3.4)
```

8.8 Makefile.in + configure (Paket mit einem Programm)

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

This is an enhanced upstream source example for Section 8.7. This also links to an external library, **libm**, and this source is configurable using arguments to the **configure** script, which generates the **Makefile** and **src/config.h** files.

Nehmen wir an, dass der Tarball der Originalautoren **debhello-1.5.tar.gz** ist.

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

```
$ tar -xzmf debhello-1.5.tar.gz
$ cd debhello-1.5
```



```
$ ./configure --with-math
$ make
$ make install
```

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

Laden Sie debhello-1.5.tar.gz herunter

```
$ wget http://www.example.org/download/debhello-1.5.tar.gz
...
$ tar -xzmf debhello-1.5.tar.gz
$ tree
.
|__|b''b''-b''b''-b'' debhello-1.5
|__|b'' b''|b''b''-b''b''-b'' LICENSE
|__|b'' b''|b''b''-b''b''-b'' Makefile.in
|__|b'' b''|b''b''-b''b''-b'' configure
|__|b'' b''|b''b''-b''b''-b'' data
|__|b'' b''|b'' b''|b''b''-b''b''-b'' hello.desktop
|__|b'' b''|b'' b''|b''b''-b''b''-b'' hello.png
|__|b'' b''|b''|b''b''-b''b''-b'' man
|__|b'' b''|b''|b'' b''|b''b''-b''b''-b'' hello.1
|__|b'' b''|b''|b''b''-b''b''-b'' src
|__|b'' b''|b''|b''b''-b''b''-b'' hello.c
|__|b''b''b''-b''b''-b'' debhello-1.5.tar.gz

4 directories, 8 files
```

Here, the contents of this source are as follows.

src/hello.c (v=1.5):

```
$ cat debhello-1.5/src/hello.c
#include "config.h"
#ifdef WITH_MATH
# include <math.h>
#endif
#include <stdio.h>
int
main()
{
    printf("Hello, I am " PACKAGE_AUTHOR "!\n");
#ifdef WITH_MATH
    printf("4.0 * atan(1.0) = %10f8\n", 4.0*atan(1.0));
#else
    printf("I can't do MATH!\n");
#endif
    return 0;
}
```

Makefile.in (v=1.5):

```
$ cat debhello-1.5/Makefile.in
prefix = @prefix@

all: src/hello

src/hello: src/hello.c
        $(CC) @VERBOSE@ \
            $(CPPFLAGS) \
            $(CFLAGS) \
            $(LDFLAGS) \
            -o $@ $^ \
            @LINKLIB@

install: src/hello
        install -D src/hello \
```

```

        $(DESTDIR)$(prefix)/bin/hello
install -m 644 -D data/hello.desktop \
        $(DESTDIR)$(prefix)/share/applications/hello.desktop
install -m 644 -D data/hello.png \
        $(DESTDIR)$(prefix)/share/pixmaps/hello.png
install -m 644 -D man/hello.1 \
        $(DESTDIR)$(prefix)/share/man/man1/hello.1

clean:
    -rm -f src/hello

distclean: clean

uninstall:
    -rm -f $(DESTDIR)$(prefix)/bin/hello
    -rm -f $(DESTDIR)$(prefix)/share/applications/hello.desktop
    -rm -f $(DESTDIR)$(prefix)/share/pixmaps/hello.png
    -rm -f $(DESTDIR)$(prefix)/share/man/man1/hello.1

.PHONY: all install clean distclean uninstall

```

configure (v=1.5):

```

$ cat debhello-1.5/configure
#!/bin/sh -e
# default values
PREFIX="/usr/local"
VERBOSE=""
WITH_MATH="0"
LINKLIB=""
PACKAGE_AUTHOR="John Doe"

# parse arguments
while [ "${1}" != "" ]; do
    VAR="${1%=*}" # Drop suffix =*
    VAL="${1#*=}" # Drop prefix *=
    case "${VAR}" in
        --prefix)
            PREFIX="${VAL}"
            ;;
        --verbose|-v)
            VERBOSE="-v"
            ;;
        --with-math)
            WITH_MATH="1"
            LINKLIB="-lm"
            ;;
        --author)
            PACKAGE_AUTHOR="${VAL}"
            ;;
        *)
            echo "W: Unknown argument: ${1}"
            esac
            shift
        done

# setup configured Makefile and src/config.h
sed -e "s,@prefix@,${PREFIX}," \
    -e "s,@VERBOSE@,${VERBOSE}," \
    -e "s,@LINKLIB@,${LINKLIB}," \
    <Makefile.in >Makefile
if [ "${WITH_MATH}" = 1 ]; then
echo "#define WITH_MATH" >src/config.h
else
echo "/* not defined: WITH_MATH */" >src/config.h

```

```
fi
echo "#define PACKAGE_AUTHOR \"${PACKAGE_AUTHOR}\"" >>src/config.h
```

Please note that the **configure** command replaces strings with **@...@** in **Makefile.in** to produce **Makefile** and creates **src/config.h**.

Let's package this with the **debmake** command.

```
$ cd debhello-1.5
$ debmake -e osamu@debian.org -f 'Osamu Aoki'
I: set parameters
...
I: sanity check of parameters
I: pkg="debhello", ver="1.5", rev="1"
I: *** start packaging in "debhello-1.5". ***
I: provide debhello_1.5.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.5.tar.gz debhello_1.5.orig.tar.gz
I: pwd = "/path/to/debhello-1.5"
I: parse binary package settings:
I: binary package=debhello Type=bin / Arch=any M-A=foreign
I: analyze the source tree
I: build_type = configure
I: scan source for copyright+license text and file extensions
I: 20 %, ext = in
...
```

Das Ergebnis ist ähnlich zu Section 4.6, aber nicht exakt identisch.

Let's inspect the notable template files generated.

debian/rules (Vorlagendatei, v=1.5):

```
$ cat debhello-1.5/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
#export DEB_CFLAGS_MAINT_APPEND = -Wall -pedantic
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
    dh $@
```

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

debian/rules (Betreuerversion, v=1.5):

```
$ vim debhello-1.5/debian/rules
... hack, hack, hack, ...
$ cat debhello-1.5/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
    dh $@

override_dh_auto_configure:
    dh_auto_configure -- \
        --with-math \
        --author="Osamu Aoki"
```

Es gibt eine Reihe von weiteren Vorlagendateien unter dem Verzeichnis **debian/**. Diese müssen auch aktualisiert werden.

The rest of the packaging activities are practically the same as the one in Section 4.8.

8.9 Autotools (Paket mit einem Programm)

Here is an example of creating a simple Debian package from a simple C source program using Autotools = Autoconf and Automake (**Makefile.am** and **configure.ac**) as its build system. See Section 5.18.1.

This source usually comes with the upstream auto-generated **Makefile.in** and **configure** files, too. This source can be packaged using these files as in Section 8.8 with the help of the **autotools-dev** package.

The better alternative is to regenerate these files using the latest Autoconf and Automake packages if the upstream provided **Makefile.am** and **configure.ac** are compatible with the latest version. This is advantageous for porting to new CPU architectures, etc. This can be automated by using the “**--with autoreconf**” option for the **dh** command.

Let’s assume this upstream tarball to be **debhello-1.6.tar.gz**.

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

```
$ tar -xzmf debhello-1.6.tar.gz
$ cd debhello-1.6
$ autoreconf -ivf # optional
$ ./configure --with-math
$ make
$ make install
```

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

Laden Sie debhello-1.6.tar.gz herunter

```
$ wget http://www.example.org/download/debhello-1.6.tar.gz
...
$ tar -xzmf debhello-1.6.tar.gz
$ tree
.
├── debhello-1.6
│   ├── LICENSE
│   ├── Makefile.am
│   ├── configure.ac
│   ├── data
│   ├── hello.desktop
│   ├── hello.png
│   ├── man
│   ├── Makefile.am
│   ├── hello.1
│   └── src
│       ├── Makefile.am
│       └── hello.c
└── debhello-1.6.tar.gz

4 directories, 10 files
```

Here, the contents of this source are as follows.

src/hello.c (v=1.6):

```
$ cat debhello-1.6/src/hello.c
#include "config.h"
#ifdef WITH_MATH
# include <math.h>
#endif
#include <stdio.h>
int
main()
{
    printf("Hello, I am " PACKAGE_AUTHOR "!\n");
#ifdef WITH_MATH
    printf("4.0 * atan(1.0) = %10f8\n", 4.0*atan(1.0));
#else
    printf("I can't do MATH!\n");
#endif
    return 0;
}
```

Makefile.am (v=1.6):

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

configure.ac (v=1.6):

```
$ cat debhello-1.6/configure.ac
#                                     -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ([2.69])
AC_INIT([debhello],[2.1],[foo@example.org])
AC_CONFIG_SRCDIR([src/hello.c])
AC_CONFIG_HEADERS([config.h])
echo "Standard customization chores"
AC_CONFIG_AUX_DIR([build-aux])
AM_INIT_AUTOMAKE([foreign])
# Add #define PACKAGE_AUTHOR ... in config.h with a comment
AC_DEFINE(PACKAGE_AUTHOR, ["Osamu Aoki"], [Define PACKAGE_AUTHOR])
echo "Add --with-math option functionality to ./configure"
AC_ARG_WITH([math],
  [AS_HELP_STRING([--with-math],
    [compile with math library @<:@default=yes@:>@])],
  [],
  [with_math="yes"])
)
echo "==== withval := \"${withval}\""
echo "==== with_math := \"${with_math}\""
# m4sh if-else construct
AS_IF([test "x${with_math}" != "xno"],[
  echo "==== Check include: math.h"
  AC_CHECK_HEADER(math.h,[],[
    AC_MSG_ERROR([Couldn't find math.h.] )
  ])
  echo "==== Check library: libm"
  AC_SEARCH_LIBS(atan, [m])
  #AC_CHECK_LIB(m, atan)
  echo "==== Build with LIBS := \"${LIBS}\""
  AC_DEFINE(WITH_MATH, [1], [Build with the math library])
],[
  echo "==== Skip building with math.h."
  AH_TEMPLATE(WITH_MATH, [Build without the math library])
])
# Checks for programs.
AC_PROG_CC
AC_CONFIG_FILES([Makefile
                 man/Makefile
                 src/Makefile])
AC_OUTPUT
```

Tip

Without “**foreign**” strictness level specified in **AM_INIT_AUTOMAKE()** as above, **automake** defaults to “**gnu**” strictness level requiring several files in the top-level directory. See “3.2 Strictness” in the **automake** document.

Let's package this with the **debmake** command.

```
$ cd debhello-1.6
$ debmake -e osamu@debian.org -f 'Osamu Aoki'
I: set parameters
...
I: sanity check of parameters
I: pkg="debhello", ver="1.6", rev="1"
I: *** start packaging in "debhello-1.6". ***
I: provide debhello_1.6.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.6.tar.gz debhello_1.6.orig.tar.gz
I: pwd = "/path/to/debhello-1.6"
I: parse binary package settings:
I: binary package=debhello Type=bin / Arch=any M-A=foreign
I: analyze the source tree
I: build_type = Autotools with autoreconf
I: scan source for copyright+license text and file extensions
I: 38 %, ext = am
...
```

The result is similar to Section 8.8 but not exactly the same.

Let's inspect the notable template files generated.

debian/rules (Vorlagendatei, v=1.6):

```
$ cat debhello-1.6/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
#export DEB_CFLAGS_MAINT_APPEND = -Wall -pedantic
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
    dh $@ --with autoreconf

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

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

debian/rules (Betreuerversion, v=1.6):

```
$ vim debhello-1.6/debian/rules
... hack, hack, hack, ...
$ cat debhello-1.6/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
    dh $@ --with autoreconf

override_dh_auto_configure:
    dh_auto_configure -- \
        --with-math
```

Es gibt eine Reihe von weiteren Vorlagendateien unter dem Verzeichnis **debian/**. Diese müssen auch aktualisiert werden.

The rest of the packaging activities are practically the same as the one in Section 4.8.

8.10 CMake (Paket mit einem Programm)

Here is an example of creating a simple Debian package from a simple C source program using CMake (**CMakeLists.txt** and some files such as **config.h.in**) as its build system. See Section 5.18.2.

The **cmake** command generates the **Makefile** file based on the **CMakeLists.txt** file and its **-D** option. It also configures the file as specified in its **configure_file(...)** by replacing strings with **@...@** and changing the **#cmakedefine ...** line.

Let's assume this upstream tarball to be **debhello-1.7.tar.gz**.

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

```
$ tar -xzf debhello-1.7.tar.gz
$ cd debhello-1.7
$ mkdir obj-x86_64-linux-gnu # for out-of-tree build
$ cd obj-x86_64-linux-gnu
$ cmake ..
$ make
$ make install
```

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

Download debhello-1.7.tar.gz

```
$ wget http://www.example.org/download/debhello-1.7.tar.gz
...
$ tar -xzf debhello-1.7.tar.gz
$ tree
.
|__ debhello-1.7
|   |__ CMakeLists.txt
|   |__ LICENSE
|   |__ data
|   |__ hello.desktop
|   |__ hello.png
|   |__ man
|   |__ CMakeLists.txt
|   |__ hello.1
|   |__ src
|   |__ CMakeLists.txt
|   |__ config.h.in
|   |__ hello.c
|__ debhello-1.7.tar.gz

4 directories, 10 files
```

Here, the contents of this source are as follows.

src/hello.c (v=1.7):

```
$ cat debhello-1.7/src/hello.c
#include "config.h"
#ifdef WITH_MATH
# include <math.h>
#endif
#include <stdio.h>
int
main()
{
    printf("Hello, I am " PACKAGE_AUTHOR "!\n");
#ifdef WITH_MATH
    printf("4.0 * atan(1.0) = %10f8\n", 4.0*atan(1.0));
#else
    printf("I can't do MATH!\n");
#endif
    return 0;
}
```

src/config.h.in (v=1.7):

```
$ cat debhello-1.7/src/config.h.in
/* name of the package author */
#define PACKAGE_AUTHOR "@PACKAGE_AUTHOR@"
/* math library support */
#cmakedefine WITH_MATH
```

CMakeLists.txt (v=1.7):

```
$ cat debhello-1.7/CMakeLists.txt
cmake_minimum_required(VERSION 2.8)
project(debhello)
set(PACKAGE_AUTHOR "Osamu Aoki")
add_subdirectory(src)
add_subdirectory(man)
$ cat debhello-1.7/man/CMakeLists.txt
install(
  FILES ${CMAKE_CURRENT_SOURCE_DIR}/hello.1
  DESTINATION share/man/man1
)
$ cat debhello-1.7/src/CMakeLists.txt
# Always define HAVE_CONFIG_H
add_definitions(-DHAVE_CONFIG_H)
# Interactively define WITH_MATH
option(WITH_MATH "Build with math support" OFF)
#variable_watch(WITH_MATH)
# Generate config.h from config.h.in
configure_file(
  "${CMAKE_CURRENT_SOURCE_DIR}/config.h.in"
  "${CMAKE_CURRENT_BINARY_DIR}/config.h"
)
include_directories("${CMAKE_CURRENT_BINARY_DIR}")
add_executable(hello hello.c)
install(TARGETS hello
  RUNTIME DESTINATION bin
)
```

Let's package this with the **debmake** command.

```
$ cd debhello-1.7
$ debmake -e osamu@debian.org -f 'Osamu Aoki'
I: set parameters
...
I: sanity check of parameters
I: pkg="debhello", ver="1.7", rev="1"
I: *** start packaging in "debhello-1.7". ***
I: provide debhello_1.7.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.7.tar.gz debhello_1.7.orig.tar.gz
I: pwd = "/path/to/debhello-1.7"
I: parse binary package settings:
I: binary package=debhello Type=bin / Arch=any M-A=foreign
I: analyze the source tree
I: build_type = Cmake
I: scan source for copyright+license text and file extensions
I: 38 %, ext = text
...
```

The result is similar to Section 8.8 but not exactly the same.

Let's inspect the notable template files generated.

debian/rules (Vorlagendatei, v=1.7):

```
$ cat debhello-1.7/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1
```



```
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
#export DEB_CFLAGS_MAINT_APPEND = -Wall -pedantic
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
    dh $@

#override_dh_auto_configure:
#    dh_auto_configure -- \
#        -DCMAKE_LIBRARY_ARCHITECTURE="$(DEB_TARGET_MULTIARCH)"
```

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

```
$ cat debhello-1.7/debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: cmake, debhelper-compat (= 13)
Standards-Version: 4.5.1
Homepage: <insert the upstream URL, if relevant>
Rules-Requires-Root: no

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

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

debian/rules (Betreuerversion, v=1.7):

```
$ vim debhello-1.7/debian/rules
... hack, hack, hack, ...
$ cat debhello-1.7/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
    dh $@

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

debian/control (Betreuerversion, v=1.7):

```
$ vim debhello-1.7/debian/control
... hack, hack, hack, ...
$ cat debhello-1.7/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: cmake, debhelper-compat (= 13)
Standards-Version: 4.5.1
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)
```

Es gibt eine Reihe von weiteren Vorlagendateien unter dem Verzeichnis **debian/**. Diese müssen auch aktualisiert werden.

The rest of the packaging activities are practically the same as the one in Section 8.8.

8.11 Autotools (Paket mit mehreren Programmen)

Here is an example of creating a set of Debian binary packages including the executable package, the shared library package, the development file package, and the debug symbol package from a simple C source program using Autotools = Autoconf and Automake (which use **Makefile.am** and **configure.ac** as their input files) as its build system. See Section 5.18.1.

Let's package this in the same way as in Section 8.9.

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

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

```
$ tar -xzmf debhello-2.0.tar.gz
$ cd debhello-2.0
$ autoreconf -ivf # optional
$ ./configure --with-math
$ make
$ make install
```

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

Download debhello-2.0.tar.gz

```
$ wget http://www.example.org/download/debhello-2.0.tar.gz
...
$ tar -xzmf debhello-2.0.tar.gz
$ tree
.
b' | b' | b' | b' | b' | debhello-2.0
b' | b' | b' | b' | b' | LICENSE
b' | b' | b' | b' | b' | Makefile.am
b' | b' | b' | b' | b' | configure.ac
b' | b' | b' | b' | b' | data
b' | b' | b' | b' | b' | b' | b' | b' | b' | hello.desktop
b' | b' | b' | b' | b' | b' | b' | b' | b' | hello.png
b' | b' | b' | b' | b' | b' | b' | b' | b' | lib
b' | b' | b' | b' | b' | b' | b' | b' | b' | Makefile.am
b' | b' | b' | b' | b' | b' | b' | b' | b' | sharedlib.c
b' | b' | b' | b' | b' | b' | b' | b' | b' | sharedlib.h
b' | b' | b' | b' | b' | b' | b' | b' | b' | man
b' | b' | b' | b' | b' | b' | b' | b' | b' | Makefile.am
b' | b' | b' | b' | b' | b' | b' | b' | b' | hello.1
b' | b' | b' | b' | b' | b' | b' | b' | b' | src
b' | b' | b' | b' | b' | b' | b' | b' | b' | Makefile.am
b' | b' | b' | b' | b' | b' | b' | b' | b' | hello.c
b' | b' | b' | b' | b' | b' | b' | b' | b' | debhello-2.0.tar.gz
```

5 directories, 13 files

Here, the contents of this source are as follows.

src/hello.c (v=2.0):

```
$ cat debhello-2.0/src/hello.c
#include "config.h"
#include <stdio.h>
#include <sharedlib.h>
int
main()
```

```
{
    printf("Hello, I am " PACKAGE_AUTHOR "!\n");
    sharedlib();
    return 0;
}
```

lib/sharedlib.h und lib/sharedlib.c (v=1.6):

```
$ cat debhello-2.0/lib/sharedlib.h
int sharedlib();
$ cat debhello-2.0/lib/sharedlib.c
#include <stdio.h>
int
sharedlib()
{
    printf("This is a shared library!\n");
    return 0;
}
```

Makefile.am (v=2.0):

```
$ cat debhello-2.0/Makefile.am
# recursively process `Makefile.am` in SUBDIRS
SUBDIRS = lib src man
$ cat debhello-2.0/man/Makefile.am
# manpages (distributed in the source package)
dist_man_MANS = hello.1
$ cat debhello-2.0/lib/Makefile.am
# libtool libraries to be produced
lib_LTLIBRARIES = libsharedlib.la

# source files used for lib_LTLIBRARIES
libsharedlib_la_SOURCES = sharedlib.c

# C pre-processor flags used for lib_LTLIBRARIES
#libsharedlib_la_CPPFLAGS =

# Headers files to be installed in <prefix>/include
include_HEADERS = sharedlib.h

# Versioning Libtool Libraries with version triplets
libsharedlib_la_LDFLAGS = -version-info 1:0:0
$ cat debhello-2.0/src/Makefile.am
# program executables to be produced
bin_PROGRAMS = hello

# source files used for bin_PROGRAMS
hello_SOURCES = hello.c

# C pre-processor flags used for bin_PROGRAMS
AM_CPPFLAGS = -I$(srcdir) -I$(top_srcdir)/lib

# Extra options for the linker for hello
# hello_LDFLAGS =

# Libraries the `hello` binary to be linked
hello_LDADD = $(top_srcdir)/lib/libsharedlib.la
```

configure.ac (v=2.0):

```
$ cat debhello-2.0/configure.ac
#
# 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])
#
# Autoconf -*-
```

```

AC_CONFIG_HEADERS([config.h])
echo "Standard customization chores"
AC_CONFIG_AUX_DIR([build-aux])

AM_INIT_AUTOMAKE([foreign])

# Set default to --enable-shared --disable-static
LT_INIT([shared disable-static])

# find the libltdl sources in the libltdl sub-directory
LT_CONFIG_LTDL_DIR([libltdl])

# choose one
LTDL_INIT([recursive])
#LTDL_INIT([subproject])
#LTDL_INIT([nonrecursive])

# Add #define PACKAGE_AUTHOR ... in config.h with a comment
AC_DEFINE(PACKAGE_AUTHOR, ["Osamu Aoki"], [Define PACKAGE_AUTHOR])
# Checks for programs.
AC_PROG_CC

# only for the recursive case
AC_CONFIG_FILES([Makefile
                 lib/Makefile
                 man/Makefile
                 src/Makefile])
AC_OUTPUT

```

Let's package this with the **debmake** command into multiple packages:

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

Here, the **-b',libsharedlib1,libsharedlib-dev'** option is used to specify the generated binary packages.

```

$ cd debhello-2.0
$ debmake -b',libsharedlib1,libsharedlib-dev' -e osamu@debian.org -f 'Osamu ...
I: set parameters
...
I: sanity check of parameters
I: pkg="debhello", ver="2.0", rev="1"
I: *** start packaging in "debhello-2.0". ***
I: provide debhello_2.0.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-2.0.tar.gz debhello_2.0.orig.tar.gz
I: pwd = "/path/to/debhello-2.0"
I: parse binary package settings: ,libsharedlib1,libsharedlib-dev
I: binary package=debhello Type=bin / Arch=any M-A=foreign
I: binary package=libsharedlib1 Type=lib / Arch=any M-A=same
I: binary package=libsharedlib-dev Type=dev / Arch=any M-A=same
I: analyze the source tree
I: build_type = Autotools with autoreconf
...

```

The result is similar to Section 8.8 but with more template files.

Let's inspect the notable template files generated.

debian/rules (Vorlagendatei, v=2.0):

```

$ cat debhello-2.0/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1

```

```
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
#export DEB_CFLAGS_MAINT_APPEND = -Wall -pedantic
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
    dh $@ --with autoreconf

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

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

debian/rules (Betreuerversion, v=2.0):

```
$ vim debhello-2.0/debian/rules
... hack, hack, hack, ...
$ cat debhello-2.0/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
    dh $@ --with autoreconf

override_dh_missing:
    dh_missing -X.la
```

debian/control (Betreuerversion, v=2.0):

```
$ vim debhello-2.0/debian/control
... hack, hack, hack, ...
$ cat debhello-2.0/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper-compat (= 13), dh-autoreconf
Standards-Version: 4.5.1
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 (Betreuerversion, v=2.0):

```
$ vim debhello-2.0/debian/debhello.install
... hack, hack, hack, ...
$ cat debhello-2.0/debian/debhello.install
usr/bin/*
usr/share/*
$ vim debhello-2.0/debian/libsharedlib1.install
... hack, hack, hack, ...
$ cat debhello-2.0/debian/libsharedlib1.install
usr/lib/*/*.so.*
$ vim debhello-2.0/debian/libsharedlib-dev.install
... hack, hack, hack, ...
$ cat debhello-2.0/debian/libsharedlib-dev.install
###usr/lib/*/pkgconfig/*.pc
usr/include
usr/lib/*/*.so
```

Since this upstream source creates the proper auto-generated **Makefile**, there is no need to create **debian/install** and **debian/manpages** files.

Es gibt eine Reihe von weiteren Vorlagendateien unter dem Verzeichnis **debian/**. Diese müssen auch aktualisiert werden.

Vorlagendateien unter debian/. (v=2.0):

```
$ tree debhello-2.0/debian
debhello-2.0/debian
b'|-b'|-b'|-b'|-b' README.Debian
b'|-b'|-b'|-b'|-b' changelog
b'|-b'|-b'|-b'|-b' control
b'|-b'|-b'|-b'|-b' copyright
b'|-b'|-b'|-b'|-b' debhello.install
b'|-b'|-b'|-b'|-b' libsharedlib-dev.install
b'|-b'|-b'|-b'|-b' libsharedlib1.install
b'|-b'|-b'|-b'|-b' libsharedlib1.symbols
b'|-b'|-b'|-b'|-b' patches
b'|-b'|-b'|-b'|-b' series
b'|-b'|-b'|-b'|-b' rules
b'|-b'|-b'|-b'|-b' source
b'|-b'|-b'|-b'|-b' control
b'|-b'|-b'|-b'|-b' format
b'|-b'|-b'|-b'|-b' local-options
b'|-b'|-b'|-b'|-b' options
b'|-b'|-b'|-b'|-b' patch-header
b'|-b'|-b'|-b'|-b' tests
b'|-b'|-b'|-b'|-b' control
b'|-b'|-b'|-b'|-b' upstream
b'|-b'|-b'|-b'|-b' metadata
b'|-b'|-b'|-b'|-b' watch
```

4 directories, 18 files

The rest of the packaging activities are practically the same as the one in Section 8.8.

Here are the generated dependency list of all binary packages.

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

```
$ dpkg -f debhello-dbgsym_2.0-1_amd64.deb pre-depends depends recommends con...
Depends: debhello (= 2.0-1)
$ dpkg -f debhello_2.0-1_amd64.deb pre-depends depends recommends conflicts ...
Depends: libsharedlib1 (= 2.0-1), libc6 (>= 2.2.5)
```

```
$ dpkg -f libsharedlib-dev_2.0-1_amd64.deb pre-depends depends recommends co...
Depends: libsharedlib1 (= 2.0-1)
$ dpkg -f libsharedlib1-dbgsym_2.0-1_amd64.deb pre-depends depends recommend...
Depends: libsharedlib1 (= 2.0-1)
$ dpkg -f libsharedlib1_2.0-1_amd64.deb pre-depends depends recommends confl...
Depends: libc6 (>= 2.2.5)
```

8.12 CMake (multi-binary package)

Here is an example of creating a set of Debian binary packages including the executable package, the shared library package, the development file package, and the debug symbol package from a simple C source program using CMake (**CMakeLists.txt** and some files such as **config.h.in**) as its build system. See Section 5.18.2.

Let's assume this upstream tarball to be **debhello-2.1.tar.gz**.

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

```
$ tar -xzf debhello-2.1.tar.gz
$ cd debhello-2.1
$ mkdir obj-x86_64-linux-gnu
$ cd obj-x86_64-linux-gnu
$ cmake ..
$ make
$ make install
```

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

Download debhello-2.1.tar.gz

```
$ wget http://www.example.org/download/debhello-2.1.tar.gz
...
$ tar -xzf debhello-2.1.tar.gz
$ tree
.
b' |b'b' |b'b' |b'b' |b'b' debhello-2.1
b' |b' |b' |b'b' |b'b' |b'b' CMakeLists.txt
b' |b' |b' |b'b' |b'b' |b'b' LICENSE
b' |b' |b' |b'b' |b'b' |b'b' data
b' |b' |b' |b'b' |b'b' |b'b' hello.desktop
b' |b' |b' |b'b' |b'b' |b'b' hello.png
b' |b' |b' |b'b' |b'b' |b'b' lib
b' |b' |b' |b'b' |b'b' |b'b' CMakeLists.txt
b' |b' |b' |b'b' |b'b' |b'b' sharedlib.c
b' |b' |b' |b'b' |b'b' |b'b' sharedlib.h
b' |b' |b' |b'b' |b'b' |b'b' man
b' |b' |b' |b'b' |b'b' |b'b' CMakeLists.txt
b' |b' |b' |b'b' |b'b' |b'b' hello.1
b' |b' |b' |b'b' |b'b' |b'b' src
b' |b' |b' |b'b' |b'b' |b'b' CMakeLists.txt
b' |b' |b' |b'b' |b'b' |b'b' config.h.in
b' |b' |b' |b'b' |b'b' |b'b' hello.c
b' |b'b' |b'b' |b'b' |b'b' debhello-2.1.tar.gz
```

5 directories, 13 files

Here, the contents of this source are as follows.

src/hello.c (v=2.1):

```
$ cat debhello-2.1/src/hello.c
#include "config.h"
#include <stdio.h>
#include <sharedlib.h>
int
main()
{
    printf("Hello, I am " PACKAGE_AUTHOR "!\\n");
```

```

    sharedlib();
    return 0;
}

```

src/config.h.in (v=2.1):

```

$ cat debhello-2.1/src/config.h.in
/* name of the package author */
#define PACKAGE_AUTHOR "@PACKAGE_AUTHOR@"

```

lib/sharedlib.c and lib/sharedlib.h (v=2.1):

```

$ cat debhello-2.1/lib/sharedlib.h
int sharedlib();
$ cat debhello-2.1/lib/sharedlib.c
#include <stdio.h>
int
sharedlib()
{
    printf("This is a shared library!\n");
    return 0;
}

```

CMakeLists.txt (v=2.1):

```

$ cat debhello-2.1/CMakeLists.txt
cmake_minimum_required(VERSION 2.8)
project(debhello)
set(PACKAGE_AUTHOR "Osamu Aoki")
add_subdirectory(lib)
add_subdirectory(src)
add_subdirectory(man)
$ cat debhello-2.1/man/CMakeLists.txt
install(
  FILES ${CMAKE_CURRENT_SOURCE_DIR}/hello.1
  DESTINATION share/man/man1
)
$ cat debhello-2.1/src/CMakeLists.txt
# Always define HAVE_CONFIG_H
add_definitions(-DHAVE_CONFIG_H)
# Generate config.h from config.h.in
configure_file(
  "${CMAKE_CURRENT_SOURCE_DIR}/config.h.in"
  "${CMAKE_CURRENT_BINARY_DIR}/config.h"
)
include_directories("${CMAKE_CURRENT_BINARY_DIR}")
include_directories("${CMAKE_SOURCE_DIR}/lib")

add_executable(hello hello.c)
target_link_libraries(hello sharedlib)
install(TARGETS hello
  RUNTIME DESTINATION bin
)

```

Let's package this with the **debmake** command.

```

$ cd debhello-2.1
$ debmake -b',libsharedlib1,libsharedlib-dev' -e osamu@debian.org -f 'Osamu ...
I: set parameters
...
I: sanity check of parameters
I: pkg="debhello", ver="2.1", rev="1"
I: *** start packaging in "debhello-2.1". ***
I: provide debhello_2.1.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-2.1.tar.gz debhello_2.1.orig.tar.gz

```



```
I: pwd = "/path/to/debhello-2.1"
I: parse binary package settings: ,libsharedlib1,libsharedlib-dev
I: binary package=debhello Type=bin / Arch=any M-A=foreign
I: binary package=libsharedlib1 Type=lib / Arch=any M-A=same
I: binary package=libsharedlib-dev Type=dev / Arch=any M-A=same
I: analyze the source tree
I: build_type = Cmake
...
```

The result is similar to Section 8.8 but not exactly the same.

Let's inspect the notable template files generated.

debian/rules (template file, v=2.1):

```
$ cat debhello-2.1/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
#export DEB_CFLAGS_MAINT_APPEND = -Wall -pedantic
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
    dh $@

#override_dh_auto_configure:
#    dh_auto_configure -- \
#        -DCMAKE_LIBRARY_ARCHITECTURE="$(DEB_TARGET_MULTIARCH)"
```

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

debian/rules (maintainer version, v=2.1):

```
$ vim debhello-2.1/debian/rules
... hack, hack, hack, ...
$ cat debhello-2.1/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed
DEB_HOST_MULTIARCH ?= $(shell dpkg-architecture -qDEB_HOST_MULTIARCH)

%:
    dh $@

override_dh_auto_configure:
    dh_auto_configure -- \
        -DCMAKE_LIBRARY_ARCHITECTURE="$(DEB_HOST_MULTIARCH)"
```

debian/control (maintainer version, v=2.1):

```
$ vim debhello-2.1/debian/control
... hack, hack, hack, ...
$ cat debhello-2.1/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: cmake, debhelper-compat (= 13)
Standards-Version: 4.5.1
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):

```
$ vim debhello-2.1/debian/debhello.install
... hack, hack, hack, ...
$ cat debhello-2.1/debian/debhello.install
usr/bin/*
usr/share/*
$ vim debhello-2.1/debian/libsharedlib1.install
... hack, hack, hack, ...
$ cat debhello-2.1/debian/libsharedlib1.install
usr/lib/*/*.so.*
$ vim debhello-2.1/debian/libsharedlib-dev.install
... hack, hack, hack, ...
$ cat debhello-2.1/debian/libsharedlib-dev.install
###usr/lib/*/*pkgconfig/*.pc
usr/include
usr/lib/*/*.so
```

This upstream CMakeList.txt needs to be patched to cope with the multiarch path.

debian/patches/* (maintainer version, v=2.1):

```
... hack, hack, hack, ...
$ cat debhello-2.1/debian/libsharedlib1.symbols
libsharedlib.so.1 libsharedlib1 #MINVER#
sharedlib@Base 2.1
```

Since this upstream source creates the proper auto-generated **Makefile**, there is no need to create **debian/install** and **debian/manpages** files.

Es gibt eine Reihe von weiteren Vorlagendateien unter dem Verzeichnis **debian/**. Diese müssen auch aktualisiert werden.

Template files under debian/. (v=2.1):

[illegible]

```

b''|b''b''-b''b''-b'' libsharedlib-dev.install
b''|b''b''-b''b''-b'' libsharedlib1.install
b''|b''b''-b''b''-b'' libsharedlib1.symbols
b''|b''b''-b''b''-b'' patches
b''|b'' b''|b''b''-b''b''-b'' 000-cmake-multiarch.patch
b''|b'' b''|b''b''-b''b''-b'' series
b''|b''b''-b''b''-b'' rules
b''|b''b''-b''b''-b'' source
b''|b'' b''|b''b''-b''b''-b'' control
b''|b'' b''|b''b''-b''b''-b'' format
b''|b'' b''|b''b''-b''b''-b'' local-options
b''|b'' b''|b''b''-b''b''-b'' options
b''|b'' b''|b''b''-b''b''-b'' patch-header
b''|b''b''-b''b''-b'' tests
b''|b'' b''|b''b''-b''b''-b'' control
b''|b''b''-b''b''-b'' upstream
b''|b'' b''|b''b''-b''b''-b'' metadata
b''|b''b''-b''b''-b'' watch

```

4 directories, 19 files

The rest of the packaging activities are practically the same as the one in Section 8.8.

Here are the generated dependency list of all binary packages.

The generated dependency list of all binary packages (v=2.1):

```

$ dpkg -f debhello-dbgSYM_2.1-1_amd64.deb pre-depends depends recommends con...
Depends: debhello (= 2.1-1)
$ dpkg -f debhello_2.1-1_amd64.deb pre-depends depends recommends conflicts ...
Depends: libsharedlib1 (= 2.1-1), libc6 (>= 2.2.5)
$ dpkg -f libsharedlib-dev_2.1-1_amd64.deb pre-depends depends recommends co...
Depends: libsharedlib1 (= 2.1-1)
$ dpkg -f libsharedlib1-dbgSYM_2.1-1_amd64.deb pre-depends depends recommend...
Depends: libsharedlib1 (= 2.1-1)
$ dpkg -f libsharedlib1_2.1-1_amd64.deb pre-depends depends recommends confl...
Depends: libc6 (>= 2.2.5)

```

8.13 Internationalization

Here is an example of updating the simple upstream C source **debhello-2.0.tar.gz** presented in Section 8.11 for internationalization (i18n) and creating the updated upstream C source **debhello-2.0.tar.gz**.

In the real situation, the package should already be internationalized. So this example is educational for you to understand how this internationalization is implemented.

Tip



The routine maintainer activity for the i18n is simply to add translation po files reported to you via the Bug Tracking System (BTS) to the **po/** directory and to update the language list in the **po/LINGUAS** file.

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

Download debhello-2.0.tar.gz (i18n)

```

$ wget http://www.example.org/download/debhello-2.0.tar.gz
...
$ tar -xzmf debhello-2.0.tar.gz
$ tree
.
b''|b''b''-b''b''-b'' debhello-2.0
b''|b'' b''|b''b''-b''b''-b'' LICENSE
b''|b'' b''|b''b''-b''b''-b'' Makefile.am

```

```

b''|b''  b''|b''b''-b''b''-b'' configure.ac
b''|b''  b''|b''b''-b''b''-b'' data
b''|b''  b''|b''  b''|b''b''-b''b''-b'' hello.desktop
b''|b''  b''|b''  b''|b''b''-b''b''-b'' hello.png
b''|b''  b''|b''b''-b''b''-b'' lib
b''|b''  b''|b''  b''|b''b''-b''b''-b'' Makefile.am
b''|b''  b''|b''  b''|b''b''-b''b''-b'' sharedlib.c
b''|b''  b''|b''  b''|b''b''-b''b''-b'' sharedlib.h
b''|b''  b''|b''b''-b''b''-b'' man
b''|b''  b''|b''  b''|b''b''-b''b''-b'' Makefile.am
b''|b''  b''|b''  b''|b''b''-b''b''-b'' hello.1
b''|b''  b''|b''b''-b''b''-b'' src
b''|b''      b''|b''b''-b''b''-b'' Makefile.am
b''|b''      b''|b''b''-b''b''-b'' hello.c
b''|b''b''-b''b''-b'' debhello-2.0.tar.gz

```

5 directories, 13 files

Internationalize this source tree with the **gettextize** command and remove files auto-generated by Autotools.
run gettextize (i18n):

```

$ cd debhello-2.0
$ gettextize
Creating po/ subdirectory
Creating build-aux/ subdirectory
Copying file ABOUT-NLS
Copying file build-aux/config.rpath
Not copying intl/ directory.
Copying file po/Makefile.in.in
Copying file po/Makevars.template
Copying file po/Rules-quot
Copying file po/boldquot.sed
Copying file po/en@boldquot.header
Copying file po/en@quot.header
Copying file po/insert-header.sin
Copying file po/quot.sed
Copying file po/remove-potcdate.sin
Creating initial po/POTFILES.in
Creating po/ChangeLog
Creating directory m4
Copying file m4/gettext.m4
Copying file m4/iconv.m4
Copying file m4/lib-ld.m4
Copying file m4/lib-link.m4
Copying file m4/lib-prefix.m4
Copying file m4/nls.m4
Copying file m4/po.m4
Copying file m4/progtest.m4
Creating m4/ChangeLog
Updating Makefile.am (backup is in Makefile.am~)
Updating configure.ac (backup is in configure.ac~)
Creating ChangeLog

Please use AM_GNU_GETTEXT([external]) in order to cause autoconfiguration
to look for an external libintl.

Please create po/Makevars from the template in po/Makevars.template.
You can then remove po/Makevars.template.

Please fill po/POTFILES.in as described in the documentation.

Please run 'aclocal' to regenerate the aclocal.m4 file.
You need aclocal from GNU automake 1.9 (or newer) to do this.
Then run 'autoconf' to regenerate the configure file.

```

You will also need `config.guess` and `config.sub`, which you can get from the CV... of the 'config' project at <http://savannah.gnu.org/>. The commands to fetch th... are

```
$ wget 'http://savannah.gnu.org/cgi-bin/viewcvs/*checkout*/config/config/conf...'
$ wget 'http://savannah.gnu.org/cgi-bin/viewcvs/*checkout*/config/config/conf...
```

You might also want to copy the convenience header file `gettext.h` from the `/usr/share/gettext` directory into your package. It is a wrapper around `<libintl.h>` that implements the `configure --disable-nl...` option.

Press Return to acknowledge the previous 6 paragraphs.

```
$ rm -rf m4 build-aux *~
```

Let's check generated files under the **po/** directory.

files in po (i18n):

```
$ ls -l po
total 60
-rw-rw-r-- 1 osamu osamu 494 Jul 2 16:28 ChangeLog
-rw-rw-r-- 1 osamu osamu 17577 Jul 2 16:28 Makefile.in.in
-rw-rw-r-- 1 osamu osamu 3376 Jul 2 16:28 Makevars.template
-rw-rw-r-- 1 osamu osamu 59 Jul 2 16:28 POTFILES.in
-rw-rw-r-- 1 osamu osamu 2203 Jul 2 16:28 Rules-quot
-rw-rw-r-- 1 osamu osamu 217 Jul 2 16:28 boldquot.sed
-rw-rw-r-- 1 osamu osamu 1337 Jul 2 16:28 en@boldquot.header
-rw-rw-r-- 1 osamu osamu 1203 Jul 2 16:28 en@quot.header
-rw-rw-r-- 1 osamu osamu 672 Jul 2 16:28 insert-header.sin
-rw-rw-r-- 1 osamu osamu 153 Jul 2 16:28 quot.sed
-rw-rw-r-- 1 osamu osamu 432 Jul 2 16:28 remove-potcdate.sin
```

Let's update the **configure.ac** by adding "**AM_GNU_GETTEXT([external])**", etc..

configure.ac (i18n):

```
$ vim configure.ac
... hack, hack, hack, ...
$ cat configure.ac
#                                     -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ([2.69])
AC_INIT([dehello], [2.2], [foo@example.org])
AC_CONFIG_SRCDIR([src/hello.c])
AC_CONFIG_HEADERS([config.h])
echo "Standard customization chores"
AC_CONFIG_AUX_DIR([build-aux])

AM_INIT_AUTOMAKE([foreign])

# Set default to --enable-shared --disable-static
LT_INIT([shared disable-static])

# find the libltdl sources in the libltdl sub-directory
LT_CONFIG_LTDL_DIR([libltdl])

# choose one
LTDL_INIT([recursive])
#LTDL_INIT([subproject])
#LTDL_INIT([nonrecursive])

# Add #define PACKAGE_AUTHOR ... in config.h with a comment
AC_DEFINE(PACKAGE_AUTHOR, ["Osamu Aoki"], [Define PACKAGE_AUTHOR])
# Checks for programs.
AC_PROG_CC

# desktop file support required
```

```
AM_GNU_GETTEXT_VERSION([0.19.3])
AM_GNU_GETTEXT([external])

# only for the recursive case
AC_CONFIG_FILES([Makefile
                  po/Makefile.in
                  lib/Makefile
                  man/Makefile
                  src/Makefile])
AC_OUTPUT
```

Let's create the **po/Makevars** file from the **po/Makevars.template** file.

po/Makevars (i18n):

```
... hack, hack, hack, ...
$ diff -u po/Makevars.template po/Makevars
--- po/Makevars.template      2021-07-02 16:28:02.711724581 +0900
+++ po/Makevars 2021-07-02 16:28:02.767725251 +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:
$ rm po/Makevars.template
```

Let's update C sources for the i18n version by wrapping strings with **_(...)**.

src/hello.c (i18n):

```
... hack, hack, hack, ...
$ cat src/hello.c
#include "config.h"
#include <stdio.h>
#include <sharedlib.h>
#define _(string) gettext (string)
int
main()
{
    printf(_("Hello, I am " PACKAGE_AUTHOR "!\n"));
    sharedlib();
    return 0;
}
```

lib/sharedlib.c (i18n):

```
... hack, hack, hack, ...
$ cat lib/sharedlib.c
#include <stdio.h>
#define _(string) gettext (string)
int
sharedlib()
{
    printf(_("This is a shared library!\n"));
    return 0;
}
```

The new **gettext** (v=0.19) can handle the i18n version of the desktop file directly.

data/hello.desktop.in (I18n):

```
$ fgrep -v '[ja]=' data/hello.desktop > data/hello.desktop.in
$ rm data/hello.desktop
$ cat data/hello.desktop.in
[Desktop Entry]
Name=Hello
Comment=Greetings
Type=Application
Keywords=hello
Exec=hello
Terminal=true
Icon=hello.png
Categories=Utility;
```

Let's list the input files to extract translatable strings in **po/POTFILES.in**.

po/POTFILES.in (I18n):

```
... hack, hack, hack, ...
$ cat po/POTFILES.in
src/hello.c
lib/sharedlib.c
data/hello.desktop.in
```

Here is the updated root **Makefile.am** with **po** added to the **SUBDIRS** environment variable.

Makefile.am (I18n):

```
$ cat Makefile.am
# recursively process `Makefile.am` in SUBDIRS
SUBDIRS = po lib src man

ACLOCAL_AMFLAGS = -I m4

EXTRA_DIST = build-aux/config.rpath m4/ChangeLog
```

Let's make a translation template file, **debhhello.pot**.

po/debhhello.pot (I18n):

```
$ xgettext -f po/POTFILES.in -d debhhello -o po/debhhello.pot -k_
$ cat po/debhhello.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: 2021-07-02 16:28+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:8
#, c-format
msgid "Hello, I am "
msgstr ""

#: lib/sharedlib.c:6
```

```
#, c-format
msgid "This is a shared library!\n"
msgstr ""

#: data/hello.desktop.in:3
msgid "Hello"
msgstr ""

#: data/hello.desktop.in:4
msgid "Greetings"
msgstr ""

#: data/hello.desktop.in:6
msgid "hello"
msgstr ""
```

Lassen Sie uns eine französische Übersetzung hinzufügen.

po/LINGUAS und po/fr.po (I18n):

```
$ echo 'fr' > po/LINGUAS
$ cp po/debhello.pot po/fr.po
$ vim po/fr.po
... hack, hack, hack, ...
$ cat po/fr.po
# SOME DESCRIPTIVE TITLE.
# This file is put in the public domain.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
msgid ""
msgstr ""
"Project-Id-Version: debhello 2.2\n"
"Report-Msgid-Bugs-To: foo@example.org\n"
"POT-Creation-Date: 2015-03-01 20:22+0900\n"
"PO-Revision-Date: 2015-02-21 23:18+0900\n"
"Last-Translator: Osamu Aoki <osamu@debian.org>\n"
"Language-Team: French <LL@li.org>\n"
"Language: ja\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: src/hello.c:34
#, c-format
msgid "Hello, my name is %s!\n"
msgstr "Bonjour, je m'appelle %s!\n"

#: lib/sharedlib.c:29
#, c-format
msgid "This is a shared library!\n"
msgstr "Ceci est une bibliothèque partagée!\n"

#: data/hello.desktop.in:3
msgid "Hello"
msgstr ""

#: data/hello.desktop.in:4
msgid "Greetings"
msgstr "Salutations"

#: data/hello.desktop.in:6
msgid "hello"
msgstr ""

#: data/hello.desktop.in:9
msgid "hello.png"
```



```
msgstr ""
```

The packaging activities are practically the same as the one in Section 8.11.
You can find more i18n examples by following Section 8.14.

8.14 Details

Actual details of the examples presented and their variants can be obtained by the following.

Wie sie die Details erhalten

```
$ apt-get source debmake-doc
$ cd debmake-doc*
$ cd examples
$ view examples/README.md
```

Follow the exact instruction in `examples/README.md`.

```
$ cd examples
$ sudo apt-get install devscripts build-essentials ...
$ 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 **packge** directory
- snapshot source tree image after the **debuild** command: the **test** directory

Notable examples are:

- the Python3 script with bare distutils (v=1.8),
- the POSIX shell script with Makefile with i18n (v=3.0),
- the Python3 script with setuptools with i18n (v=3.1),
- the C source with Makefile.in + configure with i18n (v=3.2),
- the C source with Autotools with i18n with i18n (v=3.3), and
- the C source with CMake with i18n (v=3.4).

Appendix A

debmake(1)-Handbuchseite

A.1 BEZEICHNUNG

debmake - Programm zur Erstellung eines Debian-Quellpakets

A.2 ÜBERSICHT

debmake [-h] [-c | -k] [-n | -a *Paketversion.orig.tar.gz* | -d | -t] [-p *Paket*] [-u *Version*] [-r *Revision*] [-z *Erweiterung*] [-b "*Binärpaket, ...*") [-e *foo@example.org*] [-f "*Vorname Nachname*") [-i "*Bauwerkzeug*" | -j] [-l *Lizenzdatei*] [-m] [-o *file*] [-q] [-s] [-v] [-w "*Erweiterung, ...*") [-x [01234]] [-y] [-L] [-P] [-T]

A.3 BESCHREIBUNG

debmake hilft beim Bau eines Debian-Pakets aus den Originalquellen. Normalerweise erfolgt dies wie folgt:

- Der Tarball der Originalautoren wird als Datei *Paket-Version.tar.gz* heruntergeladen.
- Sie wird mit Tar entpackt, um viele Dateien unterhalb des Verzeichnisses *Paket-Version/* zu erstellen.
- Debmake wird mit dem Verzeichnis *Paket-Version/*, möglicherweise ohne Argumente, aufgerufen:
- Dateien im Verzeichnis *Paket-Version/debian/* werden manuell angepasst.
- **dpkg-buildpackage** (usually from its wrapper **debuild** or **sbuild**) is invoked in the *package-version/* directory to make Debian packages.

Stellen Sie sicher, dass die Argumente **-b**, **-f**, **-l** und **-w** durch Maskieren vor dem Einfluss der Shell geschützt sind.

A.3.1 Optionale Argumente:

-h, --help zeigt diese Hilfe und beendet das Programm.

-c, --copyright überprüft die Quellen auf Urheberrechte und Lizenztexte und beendet das Programm.

- **-c**: einfacher Ausgabestil
- **-cc**: normaler Ausgabestil (ähnlich der Datei **debian/copyright**)
- **-ccc**: Fehlersuchausgabestil

-k, --kludge vergleicht die Datei **debian/copyright** mit den Quellen und beendet das Programm.

Die Datei **debian/copyright** muss so organisiert sein, dass die allgemeinen Dateimuster vor den besonderen Ausnahmen aufgeführt sind.

- **-k**: grundlegender Ausgabestil
- **-kk**: ausführlicher Ausgabestil

-n, --native make a native Debian source package without **.orig.tar.gz**. This makes a “**3.0 (native)**” format package.

If you are thinking of packaging a Debian-specific source tree with **debian/*** in it into a native Debian package, please think otherwise. You can use the “**debmake -d -i debuild**” or “**debmake -t -i debuild**” commands to make a “**3.0 (quilt)**” format non-native Debian package. The only difference is that the **debian/changelog** file must use the non-native version scheme: *version-revision*. The non-native package is more friendly to downstream distributions.

-a Paketversion.tar.gz, --archive Paketversion.tar.gz use the upstream source tarball directly. (**-p, -u, -z**: overridden)

The upstream tarball may be specified as *package_version.orig.tar.gz* and **tar.gz**. For other cases, it may be **tar.bz2**, or **tar.xz**.

If the specified upstream tarball name contains uppercase letters, the Debian package name is generated by converting them to lowercase letters.

If the specified argument is the URL (<http://>, <https://>, or <ftp://>) to the upstream tarball, the upstream tarball is downloaded from the URL using **wget** or **curl**.

-d, --dist run the “make dist” command equivalents first to generate the upstream tarball and use it.

The “**debmake -d**” command is designed to run in the *package/* directory hosting the upstream VCS with the build system supporting the “make dist” command equivalents. (automake/autoconf, Python distutils, ...)

-t, --tar run the “tar” command to generate the upstream tarball and use it.

The “**debmake -t**” command is designed to run in the *package/* directory hosting the upstream VCS. Unless you provide the upstream version with the **-u** option or with the **debian/changelog** file, a snapshot upstream version is generated in the **0~%y%m%d%H%M** format, e.g., *0~1403012359*, from the UTC date and time. The generated tarball excludes the **debian/** directory found in the upstream VCS. (It also excludes typical VCS directories: **.git/ .hg/ .svn/ .CVS/**.)

-p Paket, --package Paket setzt den Debian-Paketnamen.

-u Version, --upstreamversion Version setzt die Paketversion der Originalautoren.

-r Revision, --revision Revision setzt die Debian-Paketrevision.

-z extension, --targz extension set the tarball type, *extension*=(**tar.gz|tar.bz2|tar.xz**). (alias: **z, b, x**)

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

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

- **bin**: C/C++ compiled ELF binary code package (any, foreign) (default, alias: **""**, i.e., *null-string*)
- **data**: Data (fonts, graphics, ...) package (all, foreign) (alias: **da**)
- **dev**: Library development package (any, same) (alias: **de**)
- **doc**: Documentation package (all, foreign) (alias: **do**)
- **lib**: Library package (any, same) (alias: **l**)
- **perl**: Perl script package (all, foreign) (alias: **pl**)
- **python3**: Python (version 3) script package (all, foreign) (alias: **py3**)
- **ruby**: Ruby script package (all, foreign) (alias: **rb**)
- **nodejs**: Node.js based JavaScript package (all, foreign) (alias: **js**)
- **script**: Shell script package (all, foreign) (alias: **sh**)

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

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

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

-e *foo@example.org*, --email *foo@example.org* set e-mail address.

The default is taken from the value of the environment variable **\$DEBEMAIL**.

-f "*firstname lastname*", --fullname "*firstname lastname*" set the fullname.

The default is taken from the value of the environment variable **\$DEBFULLNAME**.

-i "*buildtool*", --invoke "*buildtool*" invoke "*buildtool*" at the end of execution. *buildtool* may be "**dpkg-buildpackage**", "**debuild**", "**sbuild**", etc.

The default is not to execute any program.

Setting this option automatically sets the **--local** option.

-j, --judge run **dpkg-depcheck** to judge build dependencies and identify file paths. Log files are in the parent directory.

- **package.build-dep.log**: Log file for **dpkg-depcheck**.
- **package.install.log**: Log file recording files in the **debian/tmp** directory.

-l "*license_file,...*", --license "*license_file,...*" add formatted license text to the end of the **debian/copyright** file holding license scan results.

The default is to add **COPYING** and **LICENSE**, and *license_file* needs to list only the additional file names all separated by " ,".

-m, --monoarch force packages to be non-multiarch.

-o *file*, --option *file* read optional parameters from *file*. (This is not for everyday use.)

The content of *file* is sourced as the Python code at the end of **para.py**. For example, the package description can be specified by the following file.

```
para['desc'] = 'program short description'
para['desc_long'] = '''\
program long description which you wish to include.
.
Empty line is space + .
You keep going on ...
'''
```

-q, --quitearly quit early before creating files in the **debian/** directory.

-s, --spec use upstream spec (setup.py for Python, etc.) for the package description.

-v, --version show version information.

-w "*addon,...*", --with "*addon,...*" add extra arguments to the **--with** option of the **dh(1)** command as *addon* in **debian/rules**.

The *addon* values are listed all separated by " ,", e.g., "**-w python3,autoreconf**".

For Autotools based packages, **autoreconf** as *addon* to run "**autoreconf -i -v -f**" for every package building is default behavior of the **dh(1)** command.

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 **setup.py** based 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 *n*, --extra *n* generate configuration files as templates. (Please note **debian/changelog**, **debian/control**, **debian/copyright**, and **debian/rules** are bare minimum configuration files to build a Debian binary package.)

The number *n* determines which configuration templates are generated.

- **-x0**: bare minimum configuration files. (default option if any of bare minimum configuration files already exist)

- **-x1**: all **-x0** files + desirable configuration files for the single binary package. (default option for the single binary package if none of bare minimum configuration files exist)
- **-x2**: all **-x1** files + desirable configuration files for the multi binary package. (default option for the multi binary package if none of bare minimum configuration files exist)
- **-x3**: all **-x2** files + unusual configuration template files. Unusual configuration template files are generated with the extra **.ex** suffix to ease their removal. To use these as configuration files, rename their file names to ones without the **.ex** suffix.
- **-x4**: all **-x3** files + copyright file examples.

-y, --yes “force yes” for all prompts. (without option: “ask [Y/n]”; doubled option: “force no”)

-L, --local generate configuration files for the local package to fool **lintian(1)** checks.

-P, --pedantic pedantically check auto-generated files.

-T, --tutorial output tutorial comment lines in template files.

A.4 BEISPIELE

For a well behaving source, you can build a good-for-local-use installable single Debian binary package easily with one command. Test install of such a package generated in this way offers a good alternative to the traditional “**make install**” command installing into the **/usr/local** directory since the Debian package can be removed cleanly by the “**dpkg -P ...**” command. Here are some examples of how to build such test packages. (These should work in most cases. If the **-d** option does not work, try the **-t** option instead.)

Für ein typischen, mit Autoconf/Automake paketierten C-Programmquellbaum:

- **debmake -d -i debuild**

Für einen typischen Python-(Version 3-)Modulquellbaum:

- **debmake -s -d -b”:python3” -i debuild**

Für ein typisches Python-(Version 3-)Modul im Archiv *Paket-Version.tar.gz*:

- **debmake -s -a *Paket-Version.tar.gz* -b”:python3” -i debuild**

Für ein typisches Perl-Modul im Archiv *Paket-Version.tar.gz*:

- **debmake -a *Paket-Version.tar.gz* -b”:perl” -i debuild**

A.5 HELFERPAKETE

Die Paketierung könnte die Installation einiger zusätzlicher, spezialisierter Helferpakete benötigen.

- Python-(Version 3-)Programme könnten das Paket **dh-python** benötigen.
- The Autotools (Autoconf + Automake) build system may require **autotools-dev** or **dh-autoreconf** package.
- Ruby-Programme könnten das Paket **gem2deb** benötigen.
- Node.js based JavaScript programs may require the **pkg-js-tools** package.
- Java-Programme könnten das Paket **javahelper** benötigen.
- Gnome-Programme könnten das Paket **gobject-introspection** benötigen.
- usw.

A.6 CAVEAT

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

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

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

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

See the exact definition in [Chapter 5 - Control files and their fields](#) in the “Debian Policy Manual”.

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

A.7 DEBUG

Bitte berichten Sie Fehler (auf Englisch) mittels des Befehls **reportbug** gegen das Paket **debmake**.

The character set in the environment variable **\$DEBUG** determines the logging output level.

- **i**: print information
- **p**: list all global parameters
- **d**: list parsed parameters for all binary packages
- **f**: input filename for the copyright scan
- **y**: year/name split of copyright line
- **s**: line scanner for format_state
- **b**: content_state scan loop: begin-loop
- **m**: content_state scan loop: after regex match
- **e**: content_state scan loop: end-loop
- **c**: print copyright section text
- **l**: print license section text
- **a**: print author/translator section text
- **k**: sort key for debian/copyright stanza
- **n**: scan result of debian/copyright (“**debmake -k**”)

Use this as:

```
$ DEBUG=pdfbmeclak debmake ...
```

Siehe README.developer in den Quellen für weiteres.

A.8 AUTOR

Copyright © 2014-2021 Osamu Aoki <osamu@debian.org>

A.9 LIZENZ

Expat-Lizenz

A.10 SIEHE AUCH

Das Paket **debmake-doc** stellt die »Anleitung für Debian-Betreuer« in einfachem Text, HTML und PDF-Format im Verzeichnis `/usr/share/doc/debmake-doc/` zur Verfügung.

See also **dpkg-source**(1), **deb-control**(5), **debhelper**(7), **dh**(1), **dpkg-buildpackage**(1), **debuild**(1), **quilt**(1), **dpkg-depcheck**(1), **sbuidl**(1), **gbp-buildpackage**(1), and **gbp-pq**(1) manpages.