



debian

Building Debian-Packages with Git-Buildpackage

Dipl.-Ing. Mechtilde Stehmann

and

Dr. Michael Stehmann

April 6, 2025

German version



<https://ddp-team.pages.debian.net/dpb/BuildWithGBP.pdf>

English version



https://ddp-team.pages.debian.net/dpb/en_US/BuildWithGBP.pdf

Content

I. Overview	1
1. License	3
2. About the book's title	5
3. Who should read this book?	7
4. How is this book conceived?	9
4.1. Motivation	9
4.2. Under contruction	10
4.3. Tools	10
5. Conventions	13
5.1. System	13
5.2. Terminology	13
5.3. Typography	13
5.4. Source Code Representation	13
6. Quick Guide	15
6.1. Preparing the build environment	15
6.2. Using the program scripts	16
II. Preparation	17
7. Literature	21
7.1. Debian Free Software Guidelines	21
7.2. Debian Policy Manual	21
7.3. Debian Developer Reference	21
7.4. Reference for <i>Git-Buildpackage</i>	22
7.5. Manual for Debian Maintainer	22
7.6. Debian New Maintianer Guide	22
7.7. More Information	22
8. What is a Debian Package?	23
9. How to select the software to be packaged	25

10. Checking the sources	27
10.1. License verification	27
10.1.1. <i>debmake</i>	27
10.1.2. <i>licensecheck</i>	28
10.1.3. <i>scan-copyrights</i>	28
10.1.4. <i>licensing</i>	28
10.1.5. <i>cme</i>	28
10.1.6. Manually	29
10.1.7. Stumbling blocks	29
10.2. Identifying the Programming Language	29
10.3. Checking the Dependencies	29
10.3.1. Identify dependencies with <i>packages.debian.org</i>	30
10.3.2. Identify dependencies at <i>codesearch.debian.net</i>	30
10.3.3. Search for Dependencies at the Console	30
10.4. Modifications of the Source Code	30
10.4.1. Exclude complete files	31
10.4.1.1. Listing of files to be excluded	31
10.4.1.2. Case distinctions	31
10.4.1.3. Naming of packages when excluding files	32
10.4.2. Changes in individual source code files (patching)	32
10.4.2.1. Patching with <i>Quilt</i>	33
10.4.2.2. Patching in a <i>Patch-Queue-Branh</i>	33
11. Versioning of the packages	35
11.1. Package Name	35
11.2. Version Scheme	35
11.3. <i>apt</i> and <i>dpkg</i>	36
11.4. <i>uscan</i> and the file <i>debian/watch</i>	36
12. <i>dh_make</i>	39
13. Building Java Packages	41
13.1. Challenges	41
13.2. Applications and Libraries	41
13.2.1. Packaging Java programs	42
13.2.2. Packaging Java libraries	42
13.2.3. Name of the Java package	42
13.3. Dependencies for Java packages	42
13.3.1. Identify other dependencies	42
13.3.2. Identify dependencies	42
13.4. Build systems for Java packages	43
13.4.1. The build system <i>maven</i>	43
13.4.2. Packaging with <i>maven</i>	43
13.4.3. Packaging with <i>ant</i>	46
13.4.4. Packaging with <i>gradle</i>	46
13.5. Building Java Packages without build system	46

14. Building Mozilla extensions	49
14.1. Sources of extensions	49
14.2. Integration into file system	50
15. Building Python Packages	51
16. Building metapackages	53
16.1. No upstream source	53
16.2. Native Debian package	53
16.2.1. <i>debian/source/format</i>	53
16.2.2. <i>debian/control</i>	53
16.2.3. <i>debian/rules</i>	53
16.2.4. <i>debian/changelog</i>	53
17. Configuration for installation	55
17.1. <i>debconf</i>	55
17.2. <i>dbconfig-common</i>	55
18. System setup	57
18.1. Dependencies for the program script	57
18.1.1. General dependencies	57
18.1.2. Dependencies for building Java packages	58
18.1.3. Dependencies for the Mozilla extensions	58
18.2. Create the file	59
18.2.1. Path to the projects	59
18.2.2. Configuration files	59
18.2.2.1. For every project	59
18.2.2.2. For many projects	60
18.2.2.3. Fingerprint of the Maintainer-Key	60
18.2.3. <i>.bashrc</i>	60
18.3. Set up PBuilder	61
18.3.1. <i>Chroot</i>	61
18.3.2. Configuration of the Pbuilder	61
18.3.3. Set up Hooks	65
18.3.4. Hooks - Examples	66
18.3.4.1. Hook A	66
18.3.4.2. Hook B	67
18.3.4.3. Hook C	67
18.3.4.4. Hook D	68
18.3.4.5. Hook E	68
18.3.4.6. Hook F	68
18.3.4.7. Hook G	68
18.3.4.8. Hook H	69
18.3.4.9. Hook I	69
18.3.5. Alternative <i>Chroot</i> environment	69
18.4. More chroot systems	70

18.5. Set up quilt for patching	70
19. Set up Git	73
19.1. Branches	73
19.2. Mergen	73
19.3. <i>gbp.conf</i>	73
19.3.1. Sequence	74
19.3.2. Sections in the <i>gbp.conf</i>	74
19.3.3. Syntax of the options	74
19.3.4. Example	74
19.4. Git repositories on own infrastructure	76
19.4.1. Local Git repository	76
19.4.2. Own Git server	76
20. Salsa-Repositories	77
20.1. Salsa-Konto anlegen	77
20.2. Creation of a Salsa-Repository	77
20.3. Salsa-repository for the Java team	78
20.3.1. Source of the Script	78
20.3.2. Dependencies	78
20.3.3. Get access token	78
20.3.4. Register token	79
20.3.5. Call script	79
20.4. Tasks on <i>salsa.debian.org</i>	80
20.4.1. Merge Request	80
21. Packaging beyond the branch <i>Unstable</i>	81
21.1. Security-Updates	82
21.2. (Old-)Stable-Proposal	82
21.2.1. Bug report	83
21.2.2. Requirements for a patch	83
21.2.3. Dependencies for Mozilla packages	83
21.3. Stable-Backports	84
21.4. Backports-Repository	84
21.5. Experimental	84
21.6. Backporting of unfamiliar packages	84
21.7. Versioning	84
22. An email for the start	87
22.1. ITP - Intent To Package	87
22.2. RFP - Request For Package	88
22.3. ITA - Intent To Adoption	89
22.4. RFA - Request for Adoption	89
22.5. RFH - Request For Help	89
22.6. O - Orphaned	89
22.7. RFS - Request For Sponsor	89

22.8. Changes to the bug report	90
22.9. <i>usertags</i> added	90
23. Set up report bug	91
24. Autopkgtest	93
25. Reproducible builds	95
25.1. Configuration of <i>sbuild</i>	95
25.2. <i>reprotest</i>	96
26. piuparts	97
27. Overcome difficulties	99
27.1. Unfreeze a package	99
27.1.1. Request for unblocking	99
27.2. Fix release critical bugs	100
27.3. Remove package from repository	100
III. How a shell script helps to build a Debian package	101
28. First steps in the program script	103
28.1. The beginning is at the end	103
28.2. And this is what the user sees first.	104
28.3. Request project name	106
28.4. Next steps	108
29. Create a new project	109
29.1. Create configuration file	109
29.1.1. Query common variables for configuration file	111
29.1.2. Query special variables for the configuration file	120
29.1.2.1. Identifying the plugin paths	120
29.1.2.2. Variable query for Java packages	121
29.1.2.3. Variable query for Mozilla extensions	124
29.1.2.4. Variable query for Python3 packages	126
29.1.3. Saving the configuration	127
29.1.4. Example of a configuration file	130
29.2. Creating the infrastructure	130
29.2.1. Definition of paths	130
29.2.2. Creating the necessary directories	131
29.2.3. Create log file	131
29.3. Git Repositories	132
29.3.1. Does a Git repository already exist?	132
29.3.2. Selection dialog	133
29.4. Creating a new local Git repository	136
29.4.1. Add name and email address to Git repository	136

29.4.2. Repository at <i>salsa.debian.org</i>	147
29.4.2.1. Manually	147
29.4.2.2. Within the Java team	147
29.4.3. Display remote server	148
29.5. Clonen from <i>salsa.debian.org</i>	149
29.5.1. Determination of the Git branches	150
29.5.2. Git branches detect	152
29.5.3. Assign Git Branch Distribution	152
29.5.4. Add name and email address	153
29.6. Import of a Debian package	155
29.7. GnuPG Key available?	156
29.8. Starting the packaging process	157
30. Work in an created projectt	159
30.1. Load and edit configuration file	159
30.2. Modify lines in the configuration file	162
30.3. Insert line into configuration file	163
30.4. Selecting a Git branch	163
30.4.1. Check with <i>git status</i>	164
30.4.2. Error message and troubleshooting	165
30.4.3. Selection of the Debian branches	166
30.4.4. Dialog to select a branch	167
30.4.5. Change entry	168
30.4.6. Read configuration	169
30.4.7. No or only one branch exists	170
30.5. Task selection	171
31. Building a new version	175
31.1. Download changes from <i>Salsa</i>	175
31.2. Import an existing <i>patch queue</i>	177
31.2.1. First attempt to import	178
31.2.2. Another import attempt	179
31.2.3. Import successful into PQ branch	180
31.3. Tools for downloading the upstream sources.	182
31.4. Download the classic way	185
31.4.1. Archive formats	185
31.4.2. Downloading the source code	185
31.4.2.1. Download	187
31.4.2.2. Copy the source archive	189
31.4.3. Identify suffix	191
31.4.4. Detect upstream version	192
31.4.5. Exclude files from upstream archive	195
31.4.6. Create Debian source file	203
31.4.7. Verify signature	206
31.4.7.1. Download signature file	206
31.4.7.2. Signature Verification	207

31.4.8. Replace link with a copy	207
31.4.9. <i>gbp</i> Configuration File	208
31.4.10 Import to Git	218
31.5. Download and import with <i>uscan</i>	221
32. Building a new revision	225
32.1. Creating the Debian directory	226
32.2. Request: Build with <i>mh-make</i> ?	227
32.3. Display the Debian files?	228
32.4. Files in the directory <i>debian/</i>	229
32.4.1. Display the Debian files	229
32.4.2. <i>debian/source/format</i>	231
32.4.3. <i>debian/source/include.binaries</i>	232
32.4.4. <i>debian/upstream/metadata</i>	232
32.4.5. <i>debian/copyright</i>	233
32.4.6. <i>debian/control</i>	234
32.4.6.1. Fundamental structure	234
32.4.6.2. Adaptations for Java packages	236
32.4.6.3. Web-Extension-Plugin	237
32.4.6.4. Python-Plugin	237
32.4.7. <i>debian/watch</i>	238
32.4.8. <i>debian/rules</i> - Fundamental structure	242
32.4.8.1. Create the file	242
32.4.8.2. Export of variables	243
32.4.8.3. Call of the Debhelper	243
32.4.8.4. <i>debian/rules</i> - overrides	245
32.4.8.5. End of the function	245
32.4.9. <i>salsa-ci.yml</i>	246
32.4.10. <i>debian/javabuild</i>	246
32.4.11. <i><Package name>.install</i>	247
32.4.12. <i><Package name>.dirs</i>	247
32.4.13. <i><Package name .docs</i>	247
32.4.14. <i><Package name>.links</i>	248
32.4.15. <i><Package name>.desktop</i>	248
32.4.16. <i><Package name>.examples</i>	248
32.4.17. <i>README.Debian</i>	249
32.4.18. <i>README.source</i>	249
32.5. Checking the files in <i>debian/</i> with CmeFix	249
33. Making changes to upstream code	251
33.1. Working with <i>gbp pq</i>	254
33.1.1. Creating a Patch Queue Branch	254
33.1.2. Manual Editing	256
33.1.3. Troubleshooting hints	256
33.1.4. Refreshing the patch queue branch	257
33.1.5. Hints for cleaning up the patch queue	258

33.1.6. Import of existing patches	259
33.1.7. Edit source code	260
33.1.8. Export the patches	261
33.2. Using Quilt	263
33.2.1. Create patch	265
33.3. Create new patch	266
33.4. Select file for patching	267
33.4.1. Delete Patch	269
33.4.2. Restore the initial state	271
33.5. Patch selection	273
33.6. Editing Patch	274
33.7. Modify Patch	275
34. Building	277
34.1. debian/changelog	277
34.1.1. Insert version number	279
34.2. Moving the <i>gbp</i> configuration file	284
34.3. Set parameters for <i>gbp buildpackage</i>	285
34.3.1. Identify Git branch and distribution	285
34.3.2. Customize Git branch	289
34.3.3. Identify distribution	290
34.3.4. Checking the parameters	290
34.3.5. Last option to exit	291
34.3.6. Selecting the build system	293
34.4. What does <i>Sbuild</i> do?	294
34.5. Build in the <i>Sbuild</i> chroot	295
34.5.1. Creating the S-Chroot	295
34.5.2. sbuild-update	297
34.6. Build in the <i>Pbuilder</i> chroot	297
34.6.1. Create <i>base.cow</i>	298
34.6.2. git-pbuilder update	299
34.6.3. Inclusion of the *.orig archive in *.changes	300
34.6.4. Build with <i>gbp buildpackage</i>	303
34.6.5. Download dependencies	305
34.6.6. Build - compile in <i>pbuilder</i>	305
35. If building fails	307
36. Build beyond Unstable (sid)	309
37. Verifications	311
37.1. Selection of the Changes file	313
37.2. Yamllint	314
37.3. Lintian	314
37.3.1. Test with Lintian	314
37.3.2. Lintian reports	316

37.4. Uscan	316
37.5. Checking the file <i>debian/copyright</i>	319
37.6. Check with <i>debdiff</i> and <i>diffoscope</i>	319
37.6.1. <i>debdiff</i>	319
IV. Publishing	323
38. Preparation to upload the package	325
38.1. Does <i>debian/changelog</i> exist?	325
38.2. <i>debian/changelog</i> finisg	327
38.3. Building again?	332
39. Upload to Git repositories	333
39.1. Upload to salsa.debian.org	334
39.2. Upload to the own Git-Server	336
40. Upload package	337
40.1. Selection of the target repository	338
40.2. Preparation - Create signature	340
40.2.1. Use fingerprint	341
40.3. Upload with dput	342
40.4. Upload to FTP-Master	342
40.4.1. Reject a package	345
40.5. Upload to mentors.debian.net	345
40.6. Upload as <i>non-maintainer upload</i>	346
40.7. Upload to <i>people.debian.org</i>	346
40.8. Local repository	350
V. Additional components of the script	353
41. Another task	355
41.1. Create new branch	355
41.2. Entering the name or IP of your own Git server	356
41.3. Prov. AddGitServer	356
42. Head of the Script	357
42.1. Shebang	357
42.2. Copyright notice	357
42.3. Dependencies for the program script	358
42.4. Function header	358
42.5. Function for troubleshooting	358

VI. Plugins and Scripts	359
43. Java-Plugin	361
43.1. Adjustments for Java package	361
44. Maven-Plugin	363
44.1. Head of the Maven plugin	363
44.2. Notice	364
44.3. Building with Maven	364
44.4. Editing Maven files	373
44.4.1. maven.rules	373
44.4.2. maven.ignoreRules	373
44.4.3. maven.properties	374
44.4.4. PackageName.poms	375
44.4.5. README.source	375
44.5. <i>debian/rules</i> - additions for Java packages with <i>Maven</i>	376
45. Web-Extension-Plugin	377
45.1. Header for Webext-Plugins	377
45.2. Creating the <i>webext*.xpi</i> files in <i>debian/</i>	378
45.2.1. Get the name of the <i>*.xpi</i> file	378
45.2.2. <i>debian/rules</i> - Additions for Mozilla AddOns	379
45.2.3. <i>debian/control</i> - Additions for Mozilla AddOns	381
45.2.4. <i>debian/webext-*.install</i>	381
45.2.5. <i>debian/webext-*.docs</i>	381
45.2.6. <i>debian/webext-links-tb</i>	382
46. Python-Plugin	383
46.1. Customizations for Python packages	384
46.2. <i>debian/control</i> - Addition for Python packages	384
47. Scripts	387
47.1. Creation of a project within the Java team	387
47.2. Script for extracting the documentation in PDF and Epub format.	390
47.2.1. Dependencies	390
47.2.2. Procedure	390
47.3. Script for extracting the scripts.	393
47.4. <i>gitlab-ci.yml</i> für die Salsa-CI	395
VII. Anhang	A–i
Table of Figures	A–iii
Bibliography	A–vii
Index	A–xi

Part I.

Overview

1. License

The text of the book Building Debian packages with *Git-Buildpackage* by Mechtilde and Michael Stehmann is licensed under the **Creative Commons Attribution - ShareAlike 4.0 International License (CC BY-SA 4.0)**[1].

The included code is available under the terms of the **GNU General Public License Version 3** or (at your option) any later version[2].

Copyright: © 2012-2024 Mechtilde Stehmann (E-Mail: mechtilde@debian.org),
Michael Stehmann (E-Mail: michael@canzeley.de)

2. About the book's title

What a **Debian package** is, is described in chapter 8 (page 23). Let us reveal in advance: A **Debian** package is not just an archive in *deb* format containing binary files.

git-buildpackage is a **Debian** package which contains useful tools to build **Debian** packages from a **Git** repository. Some of these tools are used by the program script described in this book (chapter 28, page 103) (see chapter 18, page 57). The names of the tools in this package start with *gbp*. An important tool is *gbp buildpackage*.

The names of the tools in this package start with *gbp*. An important tool is *gbp buildpackage*.

3. Who should read this book?

The information in this book is of particular interest to users of the script. But also people who are generally interested in packaging for a distribution will find information in this book.

This book does not want to be a “Lehrbuch” for building **Debian** packages. It is more of an experience report, where the experiences have been “in Code gegossen”.

Das Buch beschreibt, wie **Debian**-Pakete auf der Basis eines **Git**-Repositoriums mit den Programmen aus dem Paket *git-buildpackage* [3] und anderen nützlichen Befehlen erstellt werden. Am Ende sollte der Leser in der Lage sein, mit der Hilfe des dargestellten Programmskripts und der Beschreibungen der einzelnen Schritte “veröffentlichungsreife” **Debian**-Pakete zu bauen.

The program script itself does **not** build **Debian** packages, but assists the user in building them. It is only an assistance program.

This book can also be used to understand problems that can arise when packaging **Debian** packages.

Packaging is basically not difficult, though there are always new challenges. Packaging is therefore fun.

4. How is this book conceived?

4.1. Motivation

What is driving us to write such a book?

To understand this, you need to know the following::

Packaging involves executing many commands in a meaningful order at the shell. In addition, many small files must be maintained and included. The smallest errors and inaccuracies usually result in the package not being built correctly. It's also time-consuming and error-prone to keep putting in the correct options."

To keep these sources of error as small as possible, these steps were combined in a shell script. In the course of the time and with each further package this script grows and is refined also always further. So far, this has already become an extensive program script.

When Mechtilde started to deal with building **Debian** packages, the question was how to document and automate these many steps. The need for documentation could not be met from the beginning with comments, neither in the individual files nor in this program script.

That's why we started early on to record our packaging steps in detail. We paid special attention to descriptions that make decisions traceable and verifiable. This makes it easier to make necessary changes..

For this reason, we have also chosen the long form as far as possible when specifying options. This facilitates readability.

So the documentation should describe the actual packaging as well as explain the script.

The **Debian** distribution is the work of many people. It consists of tens of thousands of packages. Building the packages is a major task of the package maintainers. Many package maintainers use their own scripts for this purpose. Publishing such a script is therefore a risk. If our script makes life easier for some package maintainers and introduces newcomers to package building, this venture has been worthwhile.

The described program script does not refer to a specific **Debian** package. Rather, it is intended to be used to build simple **Debian** packages in general.

It describes the steps we need to take to package the packages maintained by Mechtilde. The program script does not claim that you can use it to build a **Debian** package from any source code.

In many places, you can and must also intervene manually. The description of the processes during packaging should be of help here.

The fact that the program script presupposes the possibility of manual intervention makes it necessary for the program script to repeatedly check whether the prerequisites that the authors have assumed are actually present. This necessity unfortunately increases the size and complexity of the script and thus also the size of the book.

4.2. Under contruction

The book and the script are still “Baustellen” because new experiences keep flowing in..

The book is written in German, the interface of the program script is English. Suggestions to improve translations are welcome. As a “proof of concept”, a part of the book has already been translated into English.¹

The release of the source code is done in the `Git` repository² provided by the `Debian` project. There, the `CI/CD` (chapter 47.4, page 395) is enabled. Therefore, the book is available as `*.pdf`³ and `*.epub`⁴. You can also download the program scripts from there.⁵

4.3. Tools

“Living more comfortably with documentation” is a common phrase in our *peer group*.

Which tools can be used to create such documentation? Are these tools also available as `Debian` packages?

Mechtilde received an important hint about this at an event for the `Software Freedom Day 2012` in Cologne. There she learned about the possibilities of `noweb` [4]⁶ ⁷.

In this combination, it is possible to maintain code and its description in *one* document. Donald Knuth refers to this as “Literate Programming”⁸

Further we have dealt with the fact that `LATEX` can be used to create documents in EPUB format in addition to PDF documents. These documents can also be read with an e-book reader. (chapter 47.2, page 390)

Translating this book into another language is a particular challenge. Our tests have shown that `OmegaT`⁹ is a useful and convenient tool in this respect. The corresponding process is documented in a separate booklet¹⁰.

The bibliography is created and maintained with *jabref*. The file created in this way can be included into the `LATEX` document.

The used editor is *geany* with the plugin *geany-plugin-latext*.

Git is ingenious. Building is therefore done with the tools from the package *git-buildpackage*¹¹.

The people at `Debian` have created many useful programs that make building `Debian` packages easier and more uniform. The program script presented was therefore created

¹<https://ddp-team.pages.debian.net/dpb/BuildWithGBP.pdf>

²<https://salsa.debian.org/ddp-team/dpb>

³<https://ddp-team.pages.debian.net/dpb/BuildWithGBP.pdf>

⁴<https://ddp-team.pages.debian.net/dpb/BuildWithGBP.epub>

⁵<https://ddp-team.pages.debian.net/dpb/build-gbp.sh>

<https://ddp-team.pages.debian.net/dpb/build-gbp-maven-plugin.sh>

<https://ddp-team.pages.debian.net/dpb/build-gbp-webext-plugin.sh>

<https://ddp-team.pages.debian.net/dpb/build-gbp-python-plugin.sh>

<https://ddp-team.pages.debian.net/dpb/build-gbp-java-plugin.sh>

⁶https://sfd.koelnerlinuxtreffen.de/SFD2012/2012Robert_Stanowsky.html

⁷s.a. <https://en.wikipedia.org/wiki/Noweb> This also meant getting closer to `LATEX`

⁸<http://www.literateprogramming.com/>

⁹<https://packages.debian.org/sid/omegat>

¹⁰<https://oldmike.pages.debian.net/omegatbooklet/omegat.pdf>

¹¹<https://packages.debian.org/sid/git-buildpackage>

April 6, 2025

“on the shoulders of giants”.

It is used to call up the auxiliary programs used in an expedient order and to provide them with useful options. It should show its users the way and make their work easier. To facilitate its adaptation to the needs of its users, it is a shell script.

5. Conventions

Some notes for a better understanding of the book:

5.1. System

The book and especially the program script were created on a *64-Bit-PC* architecture. This is called *amd64* in **Debian**. Another name for this system is *x86-64*.

5.2. Terminology

A **new package** is a package, which the program script does not know yet. I.e. to this package no configuration file exists yet.

A **new version** is a new upstream version. Building a new version is followed by building a new revision.

A **new revision** denotes a new **Debian** package to be uploaded.

5.3. Typography

All program names are set in *italic*. All proper names are set in **non-proportional** type. Superscript numbers point to footnotes on the same page. Citations to an entire document point directly to the bibliography in square brackets [].

All shell command options are given in the long form, as far as this is possible. This increases the readability.

For the abbreviations used, please refer to the entries in the glossary¹.

5.4. Source Code Representation

The source code is presented in chunks (so-called code chunks). The order of these code chunks in the book often does not correspond to the order in the scripts. The fact that the order in the book and script does not have to match is an advantage of **noweb**.

¹<https://wiki.debian.org/Glossary>

6. Quick Guide

This section is designed to help users understand the steps they need to take and find their description more quickly in the book.

6.1. Preparing the build environment

The following steps are necessary to create a Debian package from the book and the scripts,

The program script itself does **not** build Debian packages, but assists the user in building them. It is only an assistance program.

1. Download from *salsa.debian.org* The source code of this book and the following two scripts can be found at <https://salsa.debian.org/ddp-team/dpb>.
2. Installing the dependencies to generate the book and build scripts locally. (Chapter 47.2.1, page 390)
3. Generate the PDF with *./create-book.sh* (chapter 47.2, page 390).
4. With *./create-buildscript.sh* the program script is generated (chapter 47.3, page 393).
5. **Alternative:** Download files from
 - <https://ddp-team.pages.debian.net/dpb/BuildWithGBP.pdf>
 - <https://ddp-team.pages.debian.net/dpb/create-book.sh>
 - <https://ddp-team.pages.debian.net/dpb/create-buildscript.sh>
6. Create symlinks to the generated scripts under */usr/local/bin* So the current scripts are also in a program path (*PATH*) and can be called everywhere without path specification.
7. Installation of all dependencies required by the program script. (Chapter 42.3, page 358)
8. Creating the required directories and files (chapter 18.2, page 59)
 - Directory for the configuration files *~/.debian_project/* (chapter 18.2.2, page 59)
 - Creating the project directory and subdirectories (chapter 18.2.1, page 59)
 - Entries in the file *~/.bashrc* (chapter 18.2.3, page 60)
 - Creating the *gbp.conf* file (chapter 19.3, page 73)
 - When using the *pbuilder* configuration (chapter 18.3.2, page 61) and hook setup (chapter 18.3.3, page 65)

6.2. Using the program scripts

1. Providing the GPG key to sign Git tags
2. Running *build-gbp.sh* (chapter 28.1, page 103)
3. Create or load the configuration file with the project name (chapter 28.3, page 106)..
4. Obtain the upstream source code (chapter 31, page 175).
5. Create or update the files in the *debian/* directory (chapter 32, page 225).
6. Make changes to the source code (chapter 33, page 251)
7. Building the **Debian** package (chapter 34, page 277).
8. Checking the quality of the built **Debian** package (chapter 37, page 311)
9. Publishing the **Debian** package (chapter 38, page 325)

The program script is modular, so that you can “aussteigen” at many points and “einsteigen” again later. It also allows manual interventions.

Before working with the program script, you should also look into the following part.

Part II.

Preparation

April 6, 2025

This part of the book starts with a little theory. Then the setup of the system including the Git repository, which is essential for *git-buildpackage*, is described.

7. Literature

Some people ask what they should have to read before starting building **Debian** packages. Others want to know where they can find helpful information. So here are recommended readings for those who (want to) build **Debian** packages.

A short introduction to the topic "Packaging for Debian" contains the **Simple Packaging Tutorial**[5]..

The following three documents are "must-reads" for any package maintainer.

7.1. Debian Free Software Guidelines

All who want to contribute to **Debian** are recommended to read the social contract. The **Debian Free Software Guidelines** (DFSG)[6] are an essential part of the social contract. They contain the conditions that a license must meet in order to be considered "free". These conditions are already significant in the selection of the software to be packaged (chapter 10, page 27).

The **Debian Free Software Guidelines** (DFSG) apply not only to software licenses, but according to clause 1 of version 1.1 of the social contract ("all components") also to the licenses of images, sounds, texts etc..

7.2. Debian Policy Manual

The **Debian Policy Manual**[7] describes the policies for the **Debian GNU/Linux** distribution. It describes the structure and contents of the **Debian** archive, various operating system design decisions, and technical requirements that each package must meet to be included in the distribution. This document is available in English only.

It is also available as **Debian** package *debian-policy*.

The **Filesystem Hierarchy Standard**[8] is an important addition to the Policy Manual.

There are also other supplementary sets of rules.¹

7.3. Debian Developer Reference

The **Developer Reference**[9] lists the procedures and resources for **Debian**-developers. The document describes how to become a new developer for the **Debian** project, the upload procedure, how to operate the bug database (bug tracking system), the mailing lists, Internet servers, etc.

This document is intended as a reference manual for all **Debian** developers. It is available as **Debian** package *developers-reference-en*. [9]

¹<https://www.debian.org/devel/index.en.html>

7.4. Reference for *Git-Buildpackage*

There are various procedures and tools for building **Debian** packages. In this book, *git-buildpackage* is used.

In addition, we therefore still recommend the *git-buildpackage*[3] reference for the build system we have chosen.

7.5. Manual for **Debian** Maintainer

This manual for the **Debian** Maintainer[10] describes how to build a **Debian** package using the *debmake* command. It is intended for normal users and aspiring package maintainers.

The focus is on the modern packaging style. Many simple examples are included.

This “Manual for **Debian** maintainers” should consider as a legacy of the “**Debian** Guide for New Package Maintainers”.

It is available as **Debian** package *debmake-doc*. [10]

7.6. **Debian** New Maintianer Guide

This mature work[11] attempts to describe building **Debian** packages in a way that is understandable to ordinary users and future developers, with working examples.

Unlike previous documents, this one builds on *debhelper* and other tools available to a developer. [11]

This document is also available in other languages and as **Debian** packages.

7.7. More Information

Other useful literature can be found at <https://www.debian.org/doc>.

On the subject of building **Debian** packages, one can find documents in other places on the Internet. However, it is important to note: “The internet does not forget anything, and somewhere there is always still an outdated documentation linked, whose obsolescence is also not recognizable due to the lack of an expiration date.” ²[12]

²About this book, chapter 2.9 in the book *Debian Package Management*

8. What is a Debian Package?

To understand the way, you should know the goal. The goal of packaging is a Debian package[5].

But what is a Debian package?

A formal answer is: A Debian package is a software package released by the Debian project.

For a package to be released by the Debian project, it must meet requirements established, written, and published for transparency by the Debian project.

A Debian package is a software package that primarily satisfies the requirements described in the Debian policy[7]. It is part of the transparency maintained by the Debian project that the exact version of the Debian policy followed in building the package is specified in the *debian/control* file. (Chapter 32.4.6, page 234)

A Debian packages is a collection of files that allows applications or libraries to be distributed through the Debian package management system. The goal of packaging is to allow the automation of installing, updating, configuring, and removing software for Debian in a consistent manner.is[5]

A Debian package is more than just an archive file containing executable code with the extension *.deb*. A Debian package consists of **four** files; three of them are archives.

- An archive file with the extension *.orig.tar.gz* resp. *.orig.tar.xz* contains the source code from which the package is built.
- Another archive file with the extension *.debian.tar.xz* contains the debian-specific files that control the build process and installation or contain additional information.
- A file contains the signatures of the archive files. This file has the extension *.dsc*. This file is signed, too.
- Finally, the executable code is in the archive with the extension *.deb*.

The Debian package system enables traceability of the path from the upstream source code to the binary Debian package. Aimed at and often achieved is a bit-accurate reproducibility of the build process[13].

How this can be checked is described in chapter 25 (page 95).

This transparency gives the user confidence that the binary package was also comprehensibly built from the published source code.

If you save software from your build system to an archive in deb format without caring about standards and rules, you don't pack a Debian package.

Several variants (releases) are offered by the Debian project at any time, namely *stable*, *testing* and *unstable*. After the release of each new *Stable* version, the previous *Stable* version is maintained for some time as *Oldstable*. Furthermore, there is still *oldoldstable* and a branch *experimental*. There, changes are tried out that could have serious effects on the overall system.

However, the *experimental* branch is not a complete distribution, but works only as an extension of *unstable* [14].

The branch *unstable* (*Sid*) is the first port of call for new programs and new versions of programs before they are integrated into *testing*.

The development of a **Debian** package usually starts in the *unstable* branch.¹

Each **Debian** package belongs to a defined development stage of the **Debian** distribution, i.e. to a specific, released version².

Each package must be aligned with its release. It must not have any dependencies on another release. Libraries may only exist in one version in a release, so that security updates are easy for the user. A **Debian** package must therefore not contain its own version of such a library.³

¹I.Concepts, chapter 2.10.1[12]

²Chapter 2.10 in [12]

³Chapter 7.1 [15]

9. How to select the software to be packaged

Often the question is asked to the project, which package is suitable for starting. To learn packaging, there is no specially created “training” package. Rather, one learns packaging by packaging **Debian** packages (“learning by doing”).

It is emphasized from the beginning that the motivation is brought along to work step by step into the subject of packaging. A good condition for it is also that it is a software, which one would like to use gladly as **Debian** package and make available to others..

Often there are also packages for which the maintainers need help or which are orphaned. This can also apply to packages that are used by themselves. To find out if and which of the installed packages are affected, there is a tool that you can install additionally as package *how-can-i-help* [16].

This installation causes *apt* to call *how-can-i-help --apt* at the end. This will then list help needing packages that have just been updated.

With *how-can-i-help --old* you can see which of the installed packages need help.

With *how-can-i-help --all* all packages are displayed for which help is needed¹.

This information can also be found at <https://www.debian.org/devel/wnpp/>. The acronym *wnpp* abbreviates *Work-Needing and Prospective Packages* and stands, mutatis mutandis, for packages on which work is needed and which are in the process of being created (prospective packages).

It is also motivating and helpful if one is active in the upstream project. In these cases, a certain sustainability of the package support is guaranteed.

For many categories of packages support teams have been formed ². If you choose a package that fits into the portfolio of such a team, you have a better chance to find support and sponsors. Support can also be found on the mailing lists of the respective teams.

Before you start packaging, you should of course check if the software is already packaged. With *apt search <name>* it can be determined whether there is already a corresponding **Debian** package. At *salsa.debian.org* you can check if there are already packaging attempts in progress.

For all questions about packaging for **Debian** there is also a special mailing list³. Just reading along this list is useful.

¹More details about *how-can-i-help* are described in Debian Package Management[12] in chapter 37.3.6.

²Lists can be found at <https://wiki.debian.org/Teams>.

³<https://lists.debian.org/debian-mentors/>

10. Checking the sources

Having selected the software to be packaged for **Debian**, the source code must now be examined in more detail. The goal of this examination is to determine that the selected software can be packaged by the package maintainer for **Debian main**. To do this, it must be verified that **all** parts of the source code conform to the **Debian Free Software Guidelines** (DFSG)[6] and are consistent with the **Debian** policy[7] .

This requires an intensive examination of **all** source code files.

10.1. License verification

In **Debian**, only Free Software as defined in the **Debian Free Software Guidelines** may be published in the **main** branch. You have to note the **Debian Free Software Guidelines** (DFSG) apply not only to software licenses, but according to clause 1 of version 1.1 of the social contract ("all components") also to licenses of images, sounds, texts, etc.

Checking this requirement has a big part in the work of a package maintainer, especially to release a new package in **Debian**.

When creating and maintaining the *debian/copyright* file, this work is then used. Thus this file is also one of the most important files of a **Debian** package. It describes the copyright situation.

This file must accurately describe the copyright and licenses of all files in a source package using a specific syntax (DEP-5).[17]

The *debian/copyright* file is checked by the FTP masters[18] when a new package is to be accepted in the **Debian** project.

To avoid overlooking entries for this purpose, at least a four-eyes-principle (peer-review)[19] applies.

The contents of the copyright file must therefore accurately reflect the licenses of all files. The license is often specified in the comments of a source file.

The entire source code must be tested from this point of view. There are some tools that help the package maintainer to perform these tests. ¹

10.1.1. *debmake*

The *debmake -cc* command is also used by the program script to create the *debian/copyright* file (Chapter 32.4.5, page 233).

In practice, it turns out that this is sometimes not sufficient. This is because the information about the licenses and the authors is not stored following a standard. Therefore the program *debmake* does not find all entries concerning the required *copyrights*. So further tests are needed .

¹<https://wiki.debian.org/CopyrightReviewTools>

10.1.2. *licensecheck*

The *licensecheck* program (from the package of the same name) is able to scan source files and reports the copyright and licenses specified in them. However, it does not summarize this information: A copyright line is generated for each file in a package.

The test was performed with the command line

```
licensecheck --check '.*' --recursive \  
            --deb-machine --lines 0 -- *
```

as detailed in the wiki².

Unfortunately, the manual page (man page) of *licensecheck* is rather unproductive. More information is provided by the command:

```
licensecheck --help
```

The output must be evaluated manually, since *licensecheck* also tries to display the copyright for so-called binary files (e.g. images, fonts).

10.1.3. *scan-copyrights*

The *scan-copyrights* program from the *libconfig-model-dpkg-perl* package can update an existing copyright file by rescanning the source.

The command line for this is:

```
scan-copyrights
```

The program can also create such a file from scratch. This is done in DEP-5 format. [17]

This program is written in Perl and uses *licensecheck*.

10.1.4. *licensing*

The *licensing* command from the *licenseutils* package can scan the source code and return found licenses with the command.

```
licensing detect *
```

It can also add license templates to new code.

10.1.5. *cme*

Another tool is *cme*. *cme* checks and/or edits the data in the configuration files. Among other things, this can be used to check whether the copyright file provided by the original author contains all the necessary information.

The *cme fix dpkg* command checks the *dpkg* files, updates obsolete parameters, and applies any fixes (Chapter 32.5, page 249).

With

²<https://wiki.debian.org/CopyrightReviewTools#licensecheck>


```
cme update dpkg-copyright
```

the licenses listed in the headers of the source code files are checked and listed.

With

```
cme check dpkg-copyright -file <path/file name>
```

data can be read from any file.

There is also a graphical user interface for this package with *libcomcnfig-model-tkui-perl*.

10.1.6. Manually

The use of these tools is sometimes not sufficient. It may happen that further authors are named somewhere in the code. This can be searched for with the following commands:

1. `grep --recursive --ignore-case "(c)" .`
2. `grep --recursive --ignore-case "copyright" .`
3. `grep --recursive --ignore-case "author" .`

10.1.7. Stumbling blocks

There are the following stumbling blocks:

1. co-delivered build dependencies
2. microcode
3. co-delivered non-free documents
4. licenses incompletely observed by the upstream author.

Such files must be excluded from the **.orig.tar.xz* to be published, if this is reasonably possible. An appropriate version designation must also be chosen. [20]. In this program script, it is intended to exclude the files using the *debian/copyright* file (chapter 31.4.5, page 195)

If the source code package contains differently licensed files, the source code package as a whole must be examined in addition to checking the licenses of the individual source code files. It must be checked whether the licenses used are compatible with each other. The *FINOS Open Source License Compliance Handbook*[21] is helpful here.

10.2. Identifying the Programming Language

Software is written in different programming languages. There are also programs written in several programming languages. Depending on the programming language, different compiler and build systems must be used.

There are several build systems for building Java packages. This is described in detail in a separate chapter (chapter 13, page 41).

10.3. Checking the Dependencies

In order to release a package in Debian main, all dependencies (including build dependencies) must already be available in Debian main[7]. This means that dependencies that are not yet available must be packaged first.

How the dependencies can be determined depends on the programming language. Hints on unfulfilled build dependencies can also be found in the build error messages.

For Java programs, there are several places where this information can be found. This is described in the corresponding chapter (chapter 13, page 41).

10.3.1. Identify dependencies with *packages.debian.org*

To determine if the required dependency is already packaged, you can start with the website *packages.debian.org*.

In the *Search package directories* section, the options *Package names only* and *Source package names* are interesting.

In the *Search the contents of packages* section, it is then the option *packages that contain files whose names contain the keyword*.

As distribution should then *all* or *Unstable* be selected.

10.3.2. Identify dependencies at *codesearch.debian.net*

Sometimes it can be helpful to look for source code in which a similar problem has been solved before.

An example is that in a package built with Maven, a certain entry in the file *debian/-maven.rules* has been done before.

Such a search can be done – as follows:

```
https://codesearch.debian.net/search?q=<SearchString>
```

10.3.3. Search for Dependencies at the Console

With *apt-file find* ³ it can be determined if needed dependencies are already packaged in Debian.

10.4. Modifications of the Source Code

The source code must be carefully checked to see if any changes need to be made to it for the Debian package. Such a need can have various causes.

The upstream release may contain parts that do not comply with the Debian policy[7] or the Debian Free Software Guidelines[6] .

Changes to the source code specifically for the Debian package can be made as follows:

- Whole files can be excluded. This is done when building a new version (chapter 31.4.5, page 195).
- Changes can be made to upstream files by *patching* (chapter 10.4.2, page 32). This is done by the program script when building a new revision (chapter 33, page 251).

³<https://manpages.debian.org/unstable/apt-file/apt-file.1.en.html>

10.4.1. Exclude complete files

If entire files need to be excluded, a source code package must be created that no longer contains these files.

Before *mk-origtargz* is called, the program script allows individual source code files to be excluded from inclusion in the *orig* archive.

The files that do not comply with the Debian policy[7] or the Debian Free Software Guidelines (DFSG)[6] must be removed.

A new version must always be built, even if no new upstream source code archive is used.

The exclusions should be documented with their rationale in the *debian/README.source* file (chapter 32.4.18, page 249).

10.4.1.1. Listing of files to be excluded

The files to be excluded can be listed either in a separate file or in the *debian/copyright* file in *DEP-5 format* [17].

Example:

```
Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: <UpstreamName>
Source: <URL to upstream source>
Files-Excluded: <all files to be excluded>
Comment: <description, why the files are excluded>
```

In the enumeration, the files to be excluded are separated by one character using spaces or line breaks with indentation. Case distinctions

10.4.1.2. Case distinctions

Often it is already known before the package is submitted to the *New Queue* that files are to be excluded. Since there is no *debian/copyright* file at this point, it is a good idea to list the files to be excluded in a separate file.

In response to the corresponding question in the program script, specify that files are to be excluded. Then the program script determines where to store the information to exclude files. This is a separate file before submitting the first package of a new project..

This is done by creating a file *<SourceName>-excluded* in the directory *PrjPath* (chapter 29.2.1, page 130).

The information in this file is to be transferred there when the *debian/copyright* file is created (chapter , page).32.4.5233

After a rejection from the *New Queue*, the package must be re-provisioned for release. Again, files whose license is not DFSG compliant may have to be excluded. The exclusion of the files already applies to the *.orig.tar.(gz | bz2 | xz) to be published. Usually, however, no other (newer) upstream version is available at this time. However, the version to be created must be larger than the previous version but smaller than the next expected upstream version..

In the case of removing files that are not DFSG compliant, this is the *+dfsg* attachment that directly follows the upstream version. If this attachment already exists, it is

incremented after *+dfsg1* (see chapter 10.4.1.3, page 32). Thus, a new tarball with a larger version label can be created with *mk-origtargz*. (Chapter 11.2, page 35)

The names of the files to be excluded are added to the then already existing file *debian/copyright* in DEP-5 format[17]. After committing these adjustments in the *debian/* directory, a new version can be built. For this the previous upstream archive is used.

In this way, a new orig tarball is created with *mk-origtargz* without the files to be excluded from the previous *.tar.gz and its contents are inserted into the existing Git repository with *gbp import-orig* (chapter 31.4.10, page 218).

If a new upstream version requires files to be excluded, or if the files to be excluded change, proceed in the same way as just explained. The copyright file must be changed and a corresponding commit must be executed. Again, when building the new version, specify that files are to be excluded.

10.4.1.3. Naming of packages when excluding files

So that the version designation of this new version is larger than that of the previous version, but smaller than the expected next upstream version, the previous version designation is provided with an appendix. This makes clear at the same time that and why the upstream code was changed.

Common additions to the version string are *+dfsg* and *+ds*⁴.

If licenses of individual files of the upstream source code do not satisfy the **Debian Free Software Guidelines** (DFSG) and therefore should not be published as sources by **Debian**, the supplement *+dfsg* is used. If other reasons require the exclusion, one takes *+ds* ("Debian Source"). [20]

This is because it is possible that initially only files that do not comply with the **Debian** policy were excluded.

If it is later determined that the uploaded original archive still contains non-free documents, this then leads to a version designation according to the pattern *<version number>+ds+dfsg*.⁵

In some cases, a plus sign (+) can cause problems especially when building Java packages.

The naming of the **Debian** packages shown also has consequences for the contents of the *debian/watch* file (chapter 11.4 page 36 and chapter 37.4 page 316).

10.4.2. Changes in individual source code files (patching)

Changes to source code files are necessary when adjustments have to be made due to different build environments. Other cases are bug fixes, especially *security patches*.

The classical method is described below with *quilt*.

Besides, this can also be done in a *patch queue* branch with *gbp pq* (chapter 10.4.2.2, page 33)

⁴<https://wiki.debian.org/DebianMentorsFaq> in section *What does "'dfsg'" or "'ds'" in the version string mean*

⁵s. Mentors-FAQ 2.6[20]

10.4.2.1. Patching with *Quilt*

Changes to source code files are made by so-called patch files. These document the original source code and the respective changes. Another file (*debian/patches/series*) controls the sequence of application of patches. This usually happens with *quilt*. *Dquilt* is a **debian**-specific adjustment for *quilt*.

For this, there is a description in the *Debian Guide for New Package Maintainers*⁶ and in the *Debian Wiki*[22]. In addition, there is a general introduction to the use of *Quilt* [23].

Chapter 18.5 (page 70) describes how to generate the **debian**-specific customization.

10.4.2.2. Patching in a *Patch-Queue-Branch*

Since a workflow with **Git** or *git-buildpackage* [3] is used to build the Debian packages, it makes sense to use *gbp pq* instead of *quilt*⁷.

The changes are then made in a separate **Git** branch.

When using *gbp pq*, *all* code changes then occur in this branch. These changes should be thematic and small-scale. Then the patches can flow more easily to upstream or be adopted from other distributions. This also makes it easier to adapt to a new upstream version.

The basic idea is that patches are imported from the **Debian** branch to the patch queue branch in such a way that one patch file in *debian/patches* corresponds to one commit on the patch queue branch at a time.

The patch queue branch is created with *gbp pq import*. (s. a. Chapter 31.2 Page 177)

The created branch is named after the branch from which it was imported. It is prefixed with *patch-queue/* to distinguish it. So, if the **Debian** packaging is done on *debian/sid* and a *gbp pq import* is done, the newly created branch is named *patch-queue/debian/sid*.

The program script allows this import before downloading a new upstream version (chapter 31.2, page 177) and as a selection option for patching (chapter 33.1, page 254).

In the patch queue branch, the familiar **Git** commands (*rebase*, *commit* and *--amend*, etc.) can be used to work on the commits. When this is done, *gbp pq export* is used to convert the commits on the *patch queue branch* back to patches in *debian/patches/* files. (Chapter 33.1, page 254).

This described workflow facilitates, for example, cherry-picking patches for stable releases, forwarding patches to new upstream releases by using *git rebase* on the patch queue branch (already applied patches are automatically detected), and reordering, dropping, and renaming patches without resorting to *quilt*. The generated patches in *debian/patches/* have all the necessary information to be forwarded to upstream, since they use a format similar to *git-format-patch*.

The main disadvantage of this workflow is the lack of history on the patch queue branch, since it is often dropped and recreated. But there is of course a full history on the **Debian** branch in the *debian/patches/* directory.

Note that *gbp pq* does not currently provide full support for DEP3 headers[24].

⁶<https://www.debian.org/doc/manuals/maint-guide/modify.en.html>

⁷section gbp.patches.html [3]

April 6, 2025

First, it will try to parse with `git-mailinfo`[25], which only supports the *From* and *Subject* fields. If neither of these are present, *gbp pq* will attempt to convert the patch from DEP3 format to a `git-mailinfo(1)` compatible format. This is done by first loading from the “author” and “subject” fields via the first line of the *Description* field. Then any additional fields (such as “Origin” and “Forwarded”) and the rest of the description (if any) are appended to the *mail body*.

11. Versioning of the packages

Both the software to be built for Debian and **Debian** packages have version names. Here, the version designation of the upstream software should be included in that of the Debian package.

The names of all Debian binary package files are structured as follows:

`<foo><><Version number>-<Debian revision number>_<Debian architecture>.deb.`

¹

11.1. Package Name

The version name is preceded by the package name.

This must be written entirely in lowercase. The version name may also contain digits. Additionally, +, -, ~ can be included.

The version designation must begin with a digit.

11.2. Version Scheme

The versioning of the packages follows a firmly defined scheme. This ensures that all tools that use this versioning can access the same scheme. It must be ensured that the system recognizes the new version designation also as larger than the previous version designation. (s.a chapter ??, page ??)

So how must the version designation of the future package be composed? After the version designation of the upstream package, debian-specific additions can be made. These may indicate if and why parts of the upstream software have been removed, or if the package has been backported to an older package. In addition, the package version name includes a **Debian** revision number. This revision number is incremented especially if the package was rebuilt because of fixes in files in the *debian/* directory.

Possible variants can be checked for their suitability with the following command:

```
dpkg --compare-versions Version1 Operant Version2 && \
&&\echo "OK"
```

As operands these are to be used: lt le eq ne gt

lt less than (<)

le less than or equal to (<=)

eq equal to (=)

ne not equal to (!=)

ge greater than or equal to (>=)

¹DebianFAQ2019, Chapter 7.3[15]

gt greater than ($>$).

The version comparison rules can be summarized as follows:²

- Strings are compared from beginning to end.
- Letters are larger than digits.
- Digits are compared as integers.
- Letters are compared in ASCII sort order.
- There are special rules for dot ($.$), plus ($+$), and tilde (\sim) characters, as follows:
Example: $0.0 < 0.5 < 0.10 < 0.99 < 1 < 1.0\sim rc1 < 1.0 < 1.0+b1 < 1.0+nmu1 < 1.1 < 2.0$.

In some cases, a plus sign ($+$) can cause problems especially when building Java packages.

The “correct” versioning plays a role if this is to be read and used by programs. These are mainly the programs *dpkg*, *apt* and *uscan*.

11.3. *apt* and *dpkg*

The correct versioning of the Debian package is then not quite easy if not built for Debian-Sid (chapter 21.7, page 84).

dpkg or *apt* must be able to correctly identify if a newer package is available in the respective release branch.

11.4. *uscan* and the file *debian/watch*

uscan is a useful tool in the Debian project. By means of *uscan* it can be checked if there is an upstream version which is more recent (i.e. higher versioned) than the current Debian package.

The *uscan* program is provided with the *devscripts* package.

Using the data contained in *debian/watch*, *uscan* checks whether newer versions are available in the original source (chapter 37.4, page 316), and can download them if necessary (chapter 31.5, page 221).

The information obtained by *uscan* is also displayed on the page of the respective maintainer

<https://qa.debian.org/developer.php?<Maintainer>@debian.org>

For this, *uscan* needs first of all a suitable *debian/watch* file. The creation of this file, which is also needed when downloading the source code with *uscan* (chapter 31.5, page 221), is supported by the program script (chapter 32.4.7, page 238).

If files are excluded and the **.orig.tar.xz* file has been named accordingly (chapter 10.4.1.3, page 32), appropriate options must be added to the *debian/watch* file. These options are *repack*, *compression*, *repacksuffix* and *dversionmangle*. Then, on the next new upstream version, the download can be performed using *uscan*. This is also true for checking with *uscan* if there is a new upstream version.

Example:

²New Maintainer Guide, chapter 2.6 [11, Kapitel 2.6] s.a. Debmake-doc, chapter 5.2

April 6, 2025

```
opts=repack,\  
compression=xz,\  
repacksuffix=+dfsg,\  
dversionmangle=s/\+dfsg// \
```


12. *dh_make*

Fortunately, there are many useful programs (utilities) that make building **Debian** packages easier and more uniform. Anyone involved in building **Debian** packages will always come across the *dh_make* tool. The package is called *dh-make* (with hyphen).

dh_make shows how a **Debian** package can potentially look like. This makes it possible to learn how to package using concrete examples. Especially instructive are the templates included in the program package.

dh_make creates many files in the *debian/* directory. For most packages, only a subset of them is needed.

The Practice shows that each time it is necessary to check which files are not needed for the specific package. Also, the generated files are incomplete and require some manual work.

To be able to use *dh_make*, some preparations have to be made. For this purpose a corresponding "working directory" is created. The downloaded source code archive is stored in it. Preferably a tar archive intended for this is used.

This tar archive is unpacked there. This can be done on the console with

```
tar --extract --auto-compress --file=<tar archiv>.tar.gz
```

dh_make must be called in the directory containing the source code. In doing so, the name of the directory containing the source code may contain characters other than lowercase letters. However, *dh_make* requires a constrained name that conforms to the <package name>-<version> scheme. *dh_make* accepts only lowercase letters for package names.

To allow *dh_make* to fill the templates correctly, the additional option *--package name* is added to *dh_make* in such cases. This can be used to force the use of the following name as the source package name. This package name must contain the version name. There must be a hyphen between package name and version name.

Thus after the change into this directory there with

```
dh_make --createorig --packagename <SourceName>
```

the source tarball in the form desired by **Debian** *<SourceName>.orig.tar.gz/xz* can be generated. Besides this, a draft of the files in the *debian/* directory is created.

The following information is requested.

Type of package = package classes single, indep, library, python with the following meaning

single A single binary package (*.deb) is created. This is the normal case

indep A binary package is created that is independent of the architecture.

library At least two binaries are created. A library package containing only the lib in /usr/lib, and another *-dev__*.deb package containing the documentation and C headers.

python

details about package The determined values for *Maintainer Name*, *Email-Address*, *Date*, *Package Name*, *Version*, *License* and *Package Type* are displayed for confirmation.

The following files are created. First the files are listed, which are needed in any case, in each case in alphabetical order. The files *README.Debian* and *README.source* can be used for information purposes. At the end follow such templates, which are used only in very few packages.

- changelog
- control
- copyright
- rules
- salsa-ci.yml.ex
- source/format
- watch.ex
- README.Debian
- README.source
- manpage.1.ex
- manpage.sgml.ex
- manpage.xml.ex
- postinst.ex
- postrm.ex
- preinst.ex
- prerm.ex
- <packagename>.cron.d.ex
- <packagename>.doc-base.EX
- <packagename>-docs.docs

The unneeded files are removed from the *debian/* directory.

The fact that the directory name must contain the respective version designation is unfavorable when using *git*. It is refrained so far from using this utility in the program script.

13. Building Java Packages

Building Java packages has special challenges. These special features are described in detail below.

For packaging Java packages, there is a special policy called **Debian-Java** policy[26].

Furthermore there is a description about packaging with the **Javatools**[27]. In it the Java helpers are described. This tutorial is also part of the **Debian** package *javahelper*.

More information can also be obtained from the **Java** FAQ[28], especially section 2.4¹.

13.1. Challenges

In a blog article[29] Hans-Christoph Steiner aptly describes the challenges for a **Debian** package with respect to Java software.

Accordingly, the **Debian-Java** team must consistently fight the *Java* standard practice of bundling all dependencies into a single **.jar* file. This means that there are no common security updates. In this model each developer must update each dependency. This works great for large organizations with staff dedicated to this task.

For the majority of **Debian** use cases, this works poorly. **Debian** delivers on the promise that people can just install *foo* appropriately to make it work, and get security updates. The user doesn't even have to know what language the program is written in, it just works.

The hope that the *Java* developer community will embrace the value of these use cases and help **Debian** by making it easier to package *Java* projects in the standard distribution method, with common dependencies that are updated independently, has so far been only rudimentarily fulfilled.

13.2. Applications and Libraries

There are two categories of Java packages: Applications and Libraries. Both Java applications and Java libraries may contain (additional) libraries in the source code which are precompiled.

If applications written in Java are to be packaged, usually Java libraries must also be provided as **Debian** packages.

Both applications and libraries are provided in **.jar* files as *Zip* archive.

Both types must basically be written as Java bytecode (**.class* files, packaged in a **.jar* archive) and with a "*Architecture: all*" (chapter 32.4.8, page 242).

If you have only one (compiled) **.jar* archive of an application or library to be packaged, a look into the *MANIFEST.MF* file of the respective *.jar* archive can help to determine the *URL* under which the source code has been published.

¹<https://www.debian.org/doc/manuals/debian-java-faq/ch2.en.html#s2.4>

13.2.1. Packaging Java programs

Java programs are intended to be executed by end users. These also require a man page, as far as they are independently executable programs.²

The package must also ensure that the correct class is used as the main class.

Additional classes in the package must be packaged into one or more **Java** archive(s) (*.jar file(s)), which can be placed in */usr/share/java* (if intended for use by other programs) or in a “private” directory in */usr/share/<package>*.

13.2.2. Packaging Java libraries

Java libraries are used to fulfill dependencies of programs. These can be build and/or runtime dependencies.

13.2.3. Name of the Java package

A special challenge is the formation of the correct name of the **Java** package.

13.3. Dependencies for Java packages

Java applications always depend on **Java** libraries as well. In addition, **Java** packages always require *default-jdk* as a dependency. Behind this is usually the currently available OpenJDK version, a free **Java** implementation with long-term support (JDK = **Java** Development Kit).

In addition, there are special dependencies that result from the build system. These are discussed below for the individual build systems.

13.3.1. Identify other dependencies

Java developers add the required build dependencies - including third-party libraries - to their source code packages in compiled form as *.jar archives.

Especially among **Java** developers it is unfortunately very common to fall back on libraries published somewhere in the **World Wide Web** and then ship them precompiled. These libraries must be replaced by packages published in **Debian**.

So, to find dependencies, search for *.jar archives in the source code.

In this case, repacking is necessary for the **Debian** maintainer to remove these files from the upstream tarball. (Chapter 33, page 251).

The excluded libraries must be made available through standalone packages.

13.3.2. Identify dependencies

If one has determined dependencies, then one must determine how these dependencies can be fulfilled. The first thing to do is to check if corresponding **Debian** packages already exist. To determine if a dependency has already been packaged in **Debian**, there are several ways to do this.

Debian-Wiki (for Maven-Packages):

²Chapter 2.3 of the **Java** Policy[26]

<https://wiki.debian.org/Java/MavenPkgs>

13.4. Build systems for Java packages

For building Java packages different build systems are used. These are *maven*, *ant* and *gradle*. This results in peculiarities that must be considered when packaging for Debian. There are also Java packages that are built without one of these build systems. The following descriptions are based on the experiences made by the authors.

13.4.1. The build system *maven*

Apache Maven is a tool for managing and understanding software projects.

A key identifying feature for the *Maven* build system is the *pom.xml* file. This contains all information about the respective software project and follows a standardized XML format.

Also essential is the default directory structure of a **Maven** project, shown below.[30]

src/main/java Application / library sources

src/main/resources Application. / library resources

src/main/filters Resource filter files

src/main/webapp Web application sources

src/test/java Test sources

src/test/resources Test resources

src/test/filters Test resource filter files

src/it Integration tests (mainly for plugins)

src/assembly Assembly- Descriptors

src/site Website

LICENSE.txt License of the project

NOTICE.txt Notes and mappings for libraries on which the project depends

README.txt Readme of the project

pom.xml Description of the project and configuration.

In most cases, it is sufficient to build the *src/main/* branch as a **Debian** package. This is especially true when building libraries that are needed as dependencies for applications.

The upstream package must meet certain requirements to be built with the **Maven** build system.

In any case, this includes at least one *pom.xml* file. This file can be located in various places within the source code. Sometimes it is provided - especially in the **Maven** repository (<https://mvnrepository.com/>) - only additionally.

However, if available, it is often better to choose **Github** or other Git repositories as source. In doing so, pay attention to the version! For some packages the source code is also published under the domain <https://gitbox.apache.org/repos/asf>.

13.4.2. Packaging with *maven*

For *maven*-based packages, the use of *maven-debian-helper*³ is recommended.

In the file *debian/rules* (chapter 44, page 363) is written.

³<https://manpages.debian.org/unstable/maven-debian-helper/index.html>

To support building with *maven* it contains the following commands:

mh_genrules(1) generates, at least partially, the *debian/rules* file

mh_lspoms(1) looks for all POM files specified in the project source code.

mh_make(1) generates the Debian package by reading the information from the Maven POM

mh_resolve_dependencies(1) resolves the dependencies and generates the *<package name>.substvars* file containing the list of dependent packages for use by *debian/control*.

There are also corresponding *man pages* for these commands.

Of these, only the *mh_make*⁴ command in the *build-gbp-maven-plugin.sh* plugin is used at first. (Chapter 44, page 363)

When using *mh_make*, *note that the environment in which mhmakes* is started should have all dependencies of the package to be built. Otherwise, any entries in the relevant files (e.g. *debian/control*) must be made up manually. (Chapter 44.4, page 373)

For this purpose, it is recommended to perform these operations in a dedicated *chroot* where all dependencies can be installed without burdening the actual host system. The setup of the same is described in chapter 18.4 (page 70).

From *mh_make* the following files are created in the *debian/* directory:

- *maven.cleanIgnoreRules*
- *maven.rules*
- *maven.ignoreRules*
- *maven.properties*
- *<Paketname>.poms*
- *maven.publishedRules*

In addition, *mh_make* also creates the (default) files.

- *changelog*
- *control*
- *copyright*
- *rules*
- *README.source*

created. This must be taken into account during further work on these files (chapter 32.4, page 229).

With *apt-file find*⁵ it can be determined whether there is a Debian package for an *artifactID*. This function is also used by *mh_make*. The *artifactID* is located in the *pom.xml* file from the upstream code.

See wiki.debian.org/Java/MavenPkgs/Unstable for a list of all *Maven* packages already packaged in Debian.⁶

mh_make also creates the file *debian/<package name>.poms* from the *pom.xml*.

In this file all *pom.xml* files occurring in the package are listed with their relative path. These can then be provided there with the listed options.

44 *<debian/JavaPackage.poms 44>≡*

⁴[urlhttps://manpages.debian.org/unstable/maven-debian-helper/mh_make.1.en.html](https://manpages.debian.org/unstable/maven-debian-helper/mh_make.1.en.html)

⁵<https://manpages.debian.org/unstable/apt-file/apt-file.1.en.html>

⁶Thanks to Thorsten Glaser and the company Tarent <https://www.tarent.de/> for this support


```

# List of POM files for the package
# Format of this file is:
# <path to pom file> [option]*
# where option can be:
# --ignore: ignore this POM and its artifact if any
# --ignore-pom: don't install the POM. To use on POM files that are created
# temporarily for certain artifacts such as Javadoc jars. [mh_install,
# mh_installpoms]
# --no-parent: remove the <parent> tag from the POM
# --package=<package>: an alternative package to use when installing this
# POM and its artifact
# --has-package-version: to indicate that the original version of the POM
# is the same as the upstream part # of the version for the package.
# --keep-elements=<elem1,elem2>: a list of XML elements to keep in the POM
# during a clean operation with mh_cleanpom or mh_installpom
# --artifact=<path>: path to the build artifact associated with this POM,
# it will be installed when using the command mh_install. [mh_install]
# --java-lib: install the jar into /usr/share/java to comply with Debian
# packaging guidelines
# --usj-name=<name>: name to use when installing the library in /usr/share/java
# --usj-version=<version>: version to use when installing the library in
# /usr/share/java
# --no-usj-versionless: don't install the versionless link in /usr/share/java
# --dest-jar=<path>: the destination for the real jar.
# It will be installed with mh_install. [mh_install]
# --classifier=<classifier>: Optional, the classifier for the jar. Empty
# by default.
# --site-xml=<location>: Optional, the location for site.xml if it needs
# to be installed. Empty by default. [mh_install]
#

```

A commonly used line is *pom.xml -no-parent -has-package-version*. This means that the `<parent>` tag is removed from the POM when building. The *-has-package-version* option specifies that the original version of the POM is the same as the upstream part of the package version.

This file can be customized within the build process of a new **Debian** revision. (Chapter 44.4, page 373).

If the *pom.xml* is in the root directory of the source code, *mh_make* creates the other files for the *debian/* directory.

In the **Maven** repository, the *pom.xml* file is often only provided separately. This file is then placed in the *debian/* directory.

Therefore, some entries in *debian/rules* (chapter 32.4.8, page 242 are required here.

After calling all *debhelper* (*dh \$@*), the *dh_auto_build* is first overridden in a *override* to perform some more preparatory configurations.

This includes copying files to the location where the build system needs them to build the desired program.

```

45 <maven-build 45>≡
    override_dh_auto_build:
        cp debian/pom.xml .
        dh_auto_build

```

More Literature

1. <https://wiki.debian.org/Java/MavenBuilder>
2. <https://wiki.debian.org/Java/MavenRepoSpec>
3. <https://wiki.ubuntu.com/JavaTeam/Specs/MavenSupportSpec>

With *apt-file find*⁷ it can be determined whether there is a *Debian* package for an *artifactID*. This function is also used by *mh_make*. The *artifactID* is located in the *pom.xml* file from the upstream code.

See wiki.debian.org/Java/MavenPkgs/Unstable for a list of all *Maven* packages already packaged in *Debian*.⁸

13.4.3. Packaging with *ant*

The *ant* build system can be recognized by the fact that there is a *build.xml* file.⁹

13.4.4. Packaging with *gradle*

13.5. Building Java Packages without build system

It is also possible to simply build a *Java* library from a directory of *.java files. This is useful when only one or a few individual *Java* classes are needed from a larger program package to satisfy build dependencies of other packages.

Thereby there is no build system for such a *Java* directory like the still to be described *ant* (chapter 13.4.3, page 46) or *maven* (chapter 13.4.1, page 43).

```
46 <java-builds 46>≡
    export DH_VERBOSE=1
    export DH_OPTIONS=-v

    Class-Path: src/net/numericalchameleon/util/phoneticalphabets.jar
    export CLASSPATH=/usr/share/java/sugar.jar

    %:
    dh $@ --with javahelper --sourcedirectory=src/net/numericalchameleon/util/

    override_jh_build:

        javac -encoding UTF-8
        src/net/numericalchameleon/util/phoneticalphabets/*
        jh_build
        jar cvf src/phoneticalphabets.jar \
        src/net/numericalchameleon/util/phoneticalphabets/*.class

        javac -encoding UTF-8 src/net/numericalchameleon/util/spokennumbers/*
        jh_build
```

⁷<https://manpages.debian.org/unstable/apt-file/apt-file.1.en.html>

⁸Thanks to Thorsten Glaser and the company Tarent <https://www.tarent.de/> for this support

⁹<https://wiki.debian.org/Java/Packaging/Ant>

The *javahelper* package provides assistance for building with *dh*. This is highly recommended for Java packaging.

Furthermore, the file *debian/javabuild* is required. This contains two columns.

Specify the **.jar* file to be built in the first column and the source code directory where the **.java* files are located in the second column.

Example:

```
#NameOfJarFile SourceDirToPackage
```

The script allows to create this file (chapter 32.4.10, page 246).

14. Building Mozilla extensions

Packaging Mozilla extensions as Debian packages has the advantage for users and administrators that extensions can be updated together with the main application.

Debian packages are installed centrally for all system users. The updates are also done with the distribution's package management system. It requires then only a tool for this purpose.

These aspects are particularly relevant for security updates or new versions of the applications.

This has the further advantage that all users work with the same version levels.

14.1. Sources of extensions

At addons.mozilla.org you can find extensions for Firefox published by Mozilla.

For the Thunderbird the extensions are published under addons.thunderbird.net.

These are there as *.xpi* archives. Provided these are source code packages, there are not necessarily also separate source code archives. To build the *.orig.tar.xz* package from source code archives of file type *.xpi*, the *mozilla-devscripts* package is required (chapter 18.1.3, page 58).

In the case of an *.xpi* file, special entries are required in the *debian/watch* file for repackaging. The *.xpi* format is a specially specified *.zip* archive. Below is the *debian/watch* file for the Thunderbird extension Mailmindr.

```
49 <watch4xpi 49>≡
    version=4
    opts=\
    repack,compression=xz,\
    uversionmangle=s/-?([^\d.]+)/~$1/tr/A-Z/a-z/, \
    filenamemangle=s/./\v?(\d\S+)\
    .*/$1/, https://addons.thunderbird.net/en-US/thunderbird/addon/mailmindr/versions/ \ (\d.\d.\d)
```

Sometimes you can also find source code releases from people and projects on, for example, Github (github.com) or self-hosted sites. These can also be available as *.tar.gz* or *.zip* archives.

From experience, the versions that have already been published in the *addons* archive are not always promptly marked as new versions there.

14.2. Integration into file system

In order to build all desired **Mozilla** extensions in a uniform way, the required parts of the script are realized as *Webext plugin* (chapter 45, page 377). This also ensures a uniform integration into the system.

From the version (≥ 78.2) of **Thunderbirds** can be used exclusively *Mail=Extensions*. This is a break in the deployment of the respective extensions. The old (for TB $\leq 68.x$) extensions do not work anymore.

The extension must now be added to the */usr/share/webext* directory with the name (id) from the *manifest.json* file and stored as a zip archive with the extension (.xpi) (chapter 45.2.4, page 381).

For this the file *debian/rules* is extended accordingly (chapter 45.2.2, page 379). In addition, the files *<packageName>.install* (chapter 45.2.4, page 381) and *<packageName>.links* (chapter 45.2.6, page 382) must always be created so that **Thunderbird**.

15. Building Python Packages

16. Building metapackages

Metapackages are those Debian packages that depends on a group of packages. This groups packages together where it makes sense to install them together.

Metapackages have some specific features.

16.1. No upstream source

Since there is no upstream source code, a metapackage does not need a file *debian/watch*.

16.2. Native Debian package

16.2.1. *debian/source/format*

16.2.2. *debian/control*

16.2.3. *debian/rules*

16.2.4. *debian/changelog*

17. Configuration for installation

17.1. *debconf*

Debconf is a configuration management system for Debian. This is used to retrieve information for the installation from the user and then take it into account during the installation.¹

With *sudo dpkg-reconfigure debconf* *debconf* can be configured itself. This also includes the graphical user interface.

first step: <https://wiki.debian.org/debconf>
<https://wiki.debian.org/debconf>

17.2. *dbconfig-common*

Dbconfig-common provides a simple, reliable and consistent method for managing databases used by Debian packages.

¹<https://wiki.debian.org/debconf>

18. System setup

It is assumed that an installed and running **Debian** system already exists. It does not matter which branch (unstable, testing or stable) is running on it.

To sign the products of the program script, a *GPG* key of the user must be available on the system. This should usefully be the key intended for the **Debian** repository.

This also applies when a virtual machine is set up and used for building.

18.1. Dependencies for the program script

The packages required to run the program script are listed in the headers of the same (chapter 42.3, page 358).

Some of the included programs are described below:

18.1.1. General dependencies

The helpful programs listed below are regularly used by the script to build a **Debian** package:

mk-origtargz from the **devsripts** package renames the original authors' tarball, optionally changes the compression and removes unwanted files. Different options are available for this[31] (see usage in chapter 31.4.6, page 203).

gbp import-orig from package **git-buildpackage** imports a new upstream version into a Git repository (see usage in chapter 31.4.10, page 218).

dh_make from the **dh-make** package can create the required *Debian* files. Practice shows that this is very incomplete and requires some manual work. Therefore, so far we refrain from using this utility in the program script. (see chapter 12, page 39).

debmake -cc from the **debmake** package of the same name searches the source files for copyright and license texts (see chapter 10.1.1, page 27). This command is used by the program script to create the *debian/copyright* file. (see usage in chapter 32.4.5, page 233).

gbp dch is also included in the **git-buildpackage** package. In the program script, *dch* is related via *gbp dch*(see usage in chapter 34.1, page 277). Further options for *dch* can be passed to *gbp dch* by *--dch-opt=<dch-options>*.

gbp import-orig from package **git-buildpackage** <p0> imports a new upstream version into a Git repository (see usage in chapter <n0>, page <n1>).

lintian in *lintian* and **dh_lintian** in *devsripts*

uscan in *devsripts*

debsign in *devsripts* also for signing at sponsoring

dput in *dputt* resp. *dput-ng*

mh_make (Plugin)

18.1.2. Dependencies for building Java packages

The following dependency is required for building with build systems for Java packages.

58a $\langle \text{Dependencies1 } 58a \rangle \equiv$
 # gradle-debian-helper, maven-debian-helper, libmaven-bundle-plugin-java,
 $\langle \text{Dependencies5 } 58b \rangle$

18.1.3. Dependencies for the Mozilla extensions

If the Mozilla extensions source code is provided only as an **.xpi* file, the following package must be available on the machine for use by *mk-origtargz*.

58b $\langle \text{Dependencies5 } 58b \rangle \equiv$ (58a)
 # mozilla-devscripts, zip

mozilla-devscripts

zip Description like zip for the Mozilla extension.

18.2. Create the file

18.2.1. Path to the projects

It makes sense to create the packaging projects each as subdirectories of a project directory. An example of this project directory is:

```
~/Projekte/Git/01_Salsa
```

This path is deposited in each configuration file in the variable *ProjectPath*. Furthermore, this can be included in the variable *DefaultProjectPath* for many projects as a default value in the file *~/debian_project/DefaultValues* (chapter 18.2.2.2, page 60)..

18.2.2. Configuration files

18.2.2.1. For every project

For each project an own configuration file is created. This is stored in the home directory of the user in the directory *./debian_project/* as file *<project name>*. Project-specific information is stored in it.

The configuration file is technically a shell script, which is created and loaded by the program script. This Shell script contains variables, to which there values are assigned. The variables provided with values in the configuration file can be used by loading from the program script.

Furthermore, the file contains comments. These can be created by the user as desired. However, some of the comments are created by the program script. These contain a property-value pair, which is structured like the assignment of a value to a variable. The technical background is that the property name contains characters that are not allowed to be used in the identifier of a variable.

If the configuration file exists, it is loaded first. (Chapter 30.1, page 159. One can edit it then, if necessary. OtherwiseExists the configuration file, so it is loaded first. (Chapter 1s, this file is created by the script. (Chapter 29.1, page 109)

The following information is stored there:

```
# !/usr/bin/bash
# ConfigFile for <OrigName>
## General parameters
SourceName
PackName
ProjectPath = ~/Projekte/Git/01_Salsa
SalsaName
Java-Package
    SalsaName = java-team/<SourceName>.git
JavaFlag
MavenPluginFlag
MavenPluginPath
```

```

Maintainer =Maintainer=Debian_Java_Maintainers<u47> __@lt@pkg-java-main-
    tainers@lists.alioth.debian.org@gt@
Uploader =Mechtilde_Stehmann __@lt@mechtilde@debian.org@gt@
Web-Extension-Packages
SalsaName = webext-team/
WebextFlag
Maintainer
Uploader
Python-Packages SalsaName = python-team/packages/
PythonFlag
Maintainer
Uploader
DefaultBranch
RecentBranch = debian/sid
RecentUpstreamSuffix = .tar.gz
RecentRepackSuffix = +dfsg | +ds
master_Dist
DownloadUrl
DownloadZip
Maintainer Mechtilde Stehmann <mechtilde@debian.org>
ExcludeFile

```

18.2.2.2. For many projects

In the `~/.debian_project/DefaultValues` file, variables are assigned values that apply to many projects.

```

60 <DefaultValues 60>≡
    #!/usr/bin/bash

    HOME=/home/<user>/ DefaultProjectPath=${HOME}/Projekte/Git/01_Salsa

```

18.2.2.3. Fingerprint of the Maintainer-Key

In the directory `~/.debian_project/` there is also a file *fingerprint* which contains the fingerprint of the maintainer key. This key is used to sign the packets. (Chapter ??, page ??)

18.2.3. .bashrc

The following entries must be included in the *.bashrc* file:

```

DEBFULLNAME="<Fullname of the Maintainer>"
DEBEMAIL="<Email addressof the Maintainers>"
export DEBEMAIL DEBFULLNAME

```

Various Debian tools recognize your email address and name using the shell environment variables `$DEBEMAIL` and `$DEBFULLNAME`.¹

These entries are read by the *DEBValues* function. (Kapitel 29.4.1, Seite 136)

Also for *dquilt* the file *.bashrc* has to be added. (Chapter 18.5, page 70)

¹Chapter 3.1 in [11]

18.3. Set up PBuilder

Gbp buildpackage uses *cowbuilder*. The *cowbuilder* command is a wrapper for *pbuilder*, which allows the use of a *pbuilder*-like interface in a *cowdancer* environment.

18.3.1. Chroot

The *pbuilder* package contains programs to build and maintain a *chroot* environment.

Chroot means *Change root*

It is good practice to build Debian packages in a *Chroot*. This is to ensure that the package can be built with the resources of the respective distribution.

When building a Debian package in this *Chroot* environment, the build dependencies are checked to see if they can be met. This can also avoid FTBFS (Failed To Build From Source) errors. Under certain circumstances, however, the following error messages may still occur.

18.3.2. Configuration of the Pbuilder

First, the directory */var/cache/pbuilder/result* must be created. This directory must be writable for the user. After that the configuration of the *pbuilder* is done.

The configuration of *pbuilder* is not completely trivial, which may make troubleshooting difficult. Therefore, we go into it in great detail here.

The default configuration is taken from the */usr/share/pbuilder/pbuilderrc* file.

```
# pbuilder defaults; edit /etc/pbuilderrc to override these and see
# pbuilderrc.5 for documentation

# Set how much output you want from pbuilder, valid values are
# E => errors only
# W => errors and warnings
# I => errors, warnings and informational
# D => all of the above and debug messages
LOGLEVEL=I
# if positive, some log messages (errors, warnings, debugs) will be colored
# auto => try automatically detection
# yes => always use colors
# no => never use colors
USECOLORS=auto

BASETGZ=/var/cache/pbuilder/base.tgz
#EXTRAPACKAGES=""
#export DEBIAN_BUILDARCH=athlon
BUILDPLACE=/var/cache/pbuilder/build
# directory inside the chroot where the build happens. See #789404
BUILDDIR=/build
# what be used as value for HOME during builds. See #441052
# The default value prevents builds to write on HOME, which is prevented on
# Debian builds too. You can set it to $BUILDDIR to get a working HOME, if
# you need to.
BUILD_HOME=/nonexistent
MIRRORSITE=http://deb.debian.org/debian
#OTHERMIRROR="deb http://www.home.com/updates/ ./"
#export http_proxy=http://your-proxy:8080/
USESHM=yes
USEPROC=yes
USEDEVFS=no
USEDEVPTS=yes
```

April 6, 2025

```
USESYSFS=yes
USENETWORK=no
USECGROUP=yes
BUILDRESULT=/var/cache/pbuilder/result/

# specifying the distribution forces the distribution on "pbuilder update"
#DISTRIBUTION=sid
# specifying the architecture passes --arch= to debootstrap; the default is
# to use the architecture of the host
#ARCHITECTURE=$(dpkg --print-architecture)
# specifying the components of the distribution, for instance to enable all
# components on Debian use "main contrib non-free" and on Ubuntu "main
# restricted universe multiverse"
COMPONENTS="main"
#specify the cache for APT
APTCACHE="/var/cache/pbuilder/aptcache/"
APTCACHEHARDLINK="yes"
REMOVEPACKAGES=""
#HOOKDIR="/usr/lib/pbuilder/hooks"
HOOKDIR=""
EATMYDATA=no
# NB: this var is private to pbuilder; ccache uses "CCACHE_DIR" instead
# CCACHEDIR="/var/cache/pbuilder/ccache"
CCACHEDIR=""

# make debconf not interact with user
export DEBIAN_FRONTEND="noninteractive"

#for pbuilder debuild
BUILDSOURCEROOTCMD="fakeroot"
PBUILDERROOTCMD="sudo -E"
# use cowbuilder for pdebuild
#PDEBUILD_PBUILDER="cowbuilder"

# Whether to generate an additional .changes file for a source-only upload,
# whilst still producing a full .changes file for any binary packages built.
SOURCE_ONLY_CHANGES=no

# additional build results to copy out of the package build area
#ADDITIONAL_BUILDRESULTS=(xunit.xml .coverage)

# command to satisfy build-dependencies; the default is an internal shell
# implementation which is relatively slow; there are two alternate
# implementations, the "experimental" implementation,
# "pbuilder-satisfydepends-experimental", which might be useful to pull
# packages from experimental or from repositories with a low APT Pin Priority,
# and the "aptitude" implementation, which will resolve build-dependencies and
# build-conflicts with aptitude which helps dealing with complex cases but does
# not support unsigned APT repositories
PBUILDERSATISFYDEPENDSCMD="/usr/lib/pbuilder/pbuilder-satisfydepends"

# Arguments for $PBUILDERSATISFYDEPENDSCMD.
# PBUILDERSATISFYDEPENDSOPT=()

# You can optionally make pbuilder accept untrusted repositories by setting
# this option to yes, but this may allow remote attackers to compromise the
# system. Better set a valid key for the signed (local) repository with
# $APTKEYRINGS (see below).
ALLOWUNTRUSTED=no

# Option to pass to apt-get always.
export APTGETOPT=()
# Option to pass to aptitude always.
export APTITUDEOPT=()
```

```
# Whether to use debdelta or not. If "yes" debdelta will be installed in the
# chroot
DEBDELTA=no

#Command-line option passed on to dpkg-buildpackage.
#DEBBUILDPTS="-IXXX -iXXX"
DEBBUILDPTS=""

#APT configuration files directory
APTCONFDIR=""

# the username and ID used by pbuilder, inside chroot. Needs fakeroot, really
BUILDUSERID=1234
BUILDUSERNAME=pbuilder

# BINDMOUNTS is a space separated list of things to mount
# inside the chroot.
BINDMOUNTS=""

# Set the debootstrap variant to 'buldd' type.
DEBOOTSTRAPOPTS=(
    '--variant=buldd'
    '--force-check-gpg'
)
# or unset it to make it not a buldd type.
# unset DEBOOTSTRAPOPTS

# Keyrings to use for package verification with apt, not used for debootstrap
# (use DEBOOTSTRAPOPTS). By default the debian-archive-keyring package inside
# the chroot is used.
APTKEYRINGS=()

# Set the PATH I am going to use inside pbuilder: default is
# "/usr/sbin:/usr/bin:/sbin:/bin"
export PATH="/usr/sbin:/usr/bin:/sbin:/bin"

# SHELL variable is used inside pbuilder by commands like 'su';
# and they need sane values
export SHELL=/usr/bin/bash

# The name of debootstrap command, you might want "cdebootstrap".
DEBOOTSTRAP="debootstrap"

# default file extension for pkgname-logfile
PKGNAME_LOGFILE_EXTENSION="_$(dpkg --print-architecture).build"

# default PKGNAME_LOGFILE
PKGNAME_LOGFILE=""

# default AUTOCLEANAPTCACHE
AUTOCLEANAPTCACHE=""

#default COMPRESSPROG
COMPRESSPROG="gzip"

# pbuilder copies some configuration files (like /etc/hosts or
# /etc/hostname)
# from the host system into the chroot. If the directory specified here
# exists and contains one of the copied files (without the leading /etc) that
# file will be copied from here instead of the system one
CONFDIR="/etc/pbuilder/conf_files"
```

These values may be overridden by values in the */etc/pbuilderrc* file if necessary.
 After a fresh install, the */etc/pbuilderrc* looks like this:

April 6, 2025

```
# this is your configuration file for pbuilder.
# the file in /usr/share/pbuilder/pbuilderrc is the default template.
# /etc/pbuilderrc is the one meant for overwriting defaults in
# the default template
#
# read pbuilderrc.5 document for notes on specific options.
MIRRORSITE=http://ftp.de.debian.org/debian/
```

I have set these up as follows:

```
# this is your configuration file for pbuilder.
# the file in /usr/share/pbuilder/pbuilderrc is the default template.
# /etc/pbuilderrc is the one meant for overwriting defaults in
# the default template
#
# read pbuilderrc.5 document for notes on specific options.
# adapt from Mechtilde - 2019-09-07 (analog wiki)
MIRRORSITE=http://ftp.de.debian.org/debian/
AUTO_DEBSIGN=${AUTO_DEBSIGN:-no}
HOOKDIR=/var/cache/pbuilder/hooks
# Codenames for Debian suites according to their alias.
# Update these when needed.
# EXPERIMENTAL_CODENAME ="experimental"
UNSTABLE_CODENAME="sid"
TESTING_CODENAME="bookworm"
STABLE_CODENAME="bullseye"
STABLE_BACKPORTS_SUITE="$STABLE_CODENAME-backports"
```

In addition to the global configuration file */etc/pbuilderrc*, a user-specific file *~/.pbuilderrc* can also be created. The contents of this file override the system-wide settings.

```
# BINDMOUNTS is a space separated list of things to mount
# inside the chroot.
BINDMOUNTS="/var/local/repository" # lokales Verzeichnis einbinden (mounten)
OTHERMIRROR="deb http://deb.debian.org/debian/ buster-backports main \
| deb [trusted=yes] file:///var/local/repository ./"
# OTHERMIRROR="$OTHERMIRROR | deb file:///var/local/repository ./"
# Fertige Pakete im lokalen Repository ablegen
# BUILDRESULT=..

# Added after reading:
# https://lists.debian.org/debian-backports/2018/09/msg00021.html

# List of Debian suites.
DEBIAN_SUITES=($UNSTABLE_CODENAME $TESTING_CODENAME \
$STABLE_CODENAME $STABLE_BACKPORTS_SUITE
"experimental" "unstable" "testing" "stable")

# Mirrors to use. Update these to your preferred mirror.
DEBIAN_MIRROR="ftp.de.debian.org"

# Added after reading https://wiki.debian.org/cowbuilder
BASEPATH="/var/cache/pbuilder/base.cow/"
```

In addition to the system-wide and user-specific configurations, package-specific configurations of the pbuilder are also required. These files can be stored in the project directory and read in with *--configfile <configuration file>*. This configuration will overwrite any existing values. Here then the information to the projects can be stored, if for these publications in the Backports branch are needed. This file is read after all other configuration files.

See also page

<https://wiki.debian.org/BuildingFormalBackports>

the section *#Advanced: Building multi-dependency packages*

This can also be added to the `~/.pbuilderrc` file if it exists. As MIRROR a common well accessible Debian mirror is specified. If available, the address of an existing local mirror can also be entered here.

For the *hook* scripts, create a directory `~/.pbuilder/`.

18.3.3. Set up Hooks

Hooks are scripts that do things at certain predefined points during the build process. With the so-called *hooks* (hooks), the process in the build chroot can also be interrupted at predefined positions to still be able to manually intervene in the process.

The *Hook* scripts are located in the `~/.pbuilder/` directory.

The name of the hook script determines at which point in the build process the *hook* is executed.

The following conversation applies:

```
X<digit><digit><whatever-else-you-want-as-name>
```

Here it is important to specify the path and the file name used under `<i0>CI/CD</i0>`.

Unfortunately, the order in which the classes are executed in the build process does not correspond to the alphabetical order.

- A** Is for the `--build` target. It is executed before the build starts. I.e. after unpacking the build system, the source code and after the build dependency has been satisfied.
- B** Executed after the build system has successfully completed the build, before the build result is copied back. - Interrupt after successful build
- C** Executed after a build failure, before cleanup. - Interrupt after failed build
- D** Runs before unpacking the source inside the chroot environment, after the chroot environment is set up. Creates `$TMP` and `$TMPDIR` if necessary. This is called before the build dependency is satisfied. Also useful for calling `apt update`. - Ability to edit `sources.list`. `<u4>[E]` Runs after `<i0>pbuilder --update</i0>` and `<i1>pbuilder --create</i1>` finishes `apt-get`'s work with the chroot, before the kernel filesystem (`/proc`) is umounted and the tarball is created from the chroot. `<u5>[F]` Is executed just before user logs in, or program starts executing, after chroot is created in `--login` or `--execute` target. `<u6>[G]` Is executed just after `debootstrap` finishes, and configuration is loaded, and `pbuilder` starts mounting `/proc` and invoking `apt install` in `--create` target. `<u7>[H]` Is executed just after chroot unpacks, mounting `proc` and any bind mount specified in `BINDMOUNTS`. It is run for each target that needs the unpacked chroot. It is useful if you want to dynamically change the chroot mount before anything starts using it. `<u8>[I]` Runs after the build system has successfully a bcompleted the build, after copying back the build results.
- A** Is for the `--build` target. It is executed before the build starts. I.e. after unpacking the build system, the source code and after the build dependency has been satisfied.
- B** Executed after the build system has successfully completed the build, before the build result is copied back. - Interrupt after successful build

- C** Executed after a build failure, before cleanup. - Interrupt after failed build
- D** Runs before unpacking the source inside the chroot environment, after the chroot environment is set up. Creates \$TMP and \$TMPDIR if necessary. This is called before the build dependency is satisfied. Also useful for calling apt update. - Ability to edit sources.list.
- E** Runs after *pbuilder --update* and *pbuilder --create* finishes apt-get's work with the chroot, before the kernel filesystem (/proc) is unmounted and the tarball is created from the chroot.
 - <u5>[F] Is executed just before user logs in, or program starts executing, after chroot is created in --login or --execute target.
 - <u6>[G] Is executed just after debootstrap finishes, and configuration is loaded, and pbuilder starts mounting /proc and invoking apt install in --create target.
 - <u7>[H] Is executed just after chroot unpacks, mounting proc and any bind mount specified in BINDMOUNTS. It is run for each target that needs the unpacked chroot. It is useful if you want to dynamically change the chroot mount before anything starts using it.
 - <u8>[I] Runs after the build system has successfully completed the build, after copying back the build results.

18.3.4. Hooks - Examples

These are all in the directory: *~/.pbuilder/*

18.3.4.1. Hook A

This hook could be named *C10shell* for example.

```
66 <Hook-A 66>≡
    #!/usr/bin/bash
    # example file to be used with --hookdir
    #
    # invoke shell before build starts.

    BUILDDIR="${BUILDDIR:-/tmp/builddd}"

    apt-get install -y "${APTGETOPT[@]}" nano less
    cd "$BUILDDIR"/*/debian/..
    echo "Hook A - the dependencies are installed. Now the build can start."
    echo "Please use CTRL-D to continue"
    /usr/bin/bash < /dev/tty > /dev/tty 2> /dev/tty
```

18.3.4.2. Hook B

This hook could be named *C10shell* for example.

```
67a <Hook-B 67a>≡
#!/usr/bin/bash
# example file to be used with --hookdir
#
# invoke shell if build fails.

BUILDDIR="${BUILDDIR:-/tmp/builddd}"

# apt-get install -y "${APTGETOPT[@]}" vim less
cd "$BUILDDIR"/*/debian/..
echo "Hook B - The build was built successfully"
echo "You can check it with ls -la ../"
/usr/bin/bash < /dev/tty > /dev/tty 2> /dev/tty
```

18.3.4.3. Hook C

Here you can install packages that are required if the build fails.

This hook could be named *C10shell* for example.

```
67b <Hook-C 67b>≡
#!/usr/bin/bash
# example file to be used with --hookdir
#
# invoke shell if build fails.

BUILDDIR="${BUILDDIR:-/tmp/builddd}"

apt-get install -y "${APTGETOPT[@]}" vim less mc unzip locate
cd "$BUILDDIR"/*/debian/..
echo "Hook C - The build wasn't built successfully"
echo "After analysing the errors you can continue with using CTRL-D"
/usr/bin/bash < /dev/tty > /dev/tty 2> /dev/tty
```

18.3.4.4. Hook D

```
68a  <Hook-D 68a>≡
      #!/usr/bin/bash
      # example file to be used with --hookdir
      #
      # invoke shell before unpacking the source
      # inside the chroot

      BUILDDIR="${BUILDDIR:-/tmp/builddd}"

      apt-get install -y "${APTGETOPT[@]}" less nano
      #cd "$BUILDDIR"/*/debian/..
      echo "Hook D -"
      echo "After unpacking the sources the dependencies"
      echo "can be downloaded and unpacked."
      echo "Please use CTRL-D to continue"
      /usr/bin/bash < /dev/tty > /dev/tty 2> /dev/tty
```

18.3.4.5. Hook E

```
68b  <Hook-E 68b>≡
      echo "Hook E"
      echo "Please use CTRL-D to continue"
      /usr/bin/bash < /dev/tty > /dev/tty 2> /dev/tty
```

18.3.4.6. Hook F

```
68c  <Hook-F 68c>≡
      echo "Hook F"
      echo "Please use CTRL-D to continue"
      /usr/bin/bash < /dev/tty > /dev/tty 2> /dev/tty
```

18.3.4.7. Hook G

```
68d  <Hook-G 68d>≡
      echo "Hook G"
      echo "Please use CTRL-D to continue"
      /usr/bin/bash < /dev/tty > /dev/tty 2> /dev/tty
```


18.3.4.8. Hook H

```

69a  <Hook-H 69a>≡
      #!/usr/bin/bash
      # example file to be used with --hookdir
      #
      # invoke shell if build fails.

      BUILDDIR="${BUILDDIR:-/tmp/builddd}"

      echo "Hook H"
      echo "Executed after preparing the chroot \n and before installing the dependencies"
      echo "Here you can include dependency from e.g a local repo for testing."
      echo "Next the source code of the package to be built is unpacked."
      echo "Please use CTRL-D to continue"
      /usr/bin/bash < /dev/tty > /dev/tty 2> /dev/tty

```

18.3.4.9. Hook I

```

69b  <Hook-I 69b>≡
      #!/usr/bin/bash
      # example file to be used with --hookdir
      #
      # invoke shell if build fails.

      BUILDDIR="${BUILDDIR:-/tmp/builddd}"

      #apt-get install -y "${APTGETOPT[@]}" vim less
      #cd "$BUILDDIR"/*/debian/..
      echo "Hook I"
      echo "Please use CTRL-D to continue"
      /usr/bin/bash < /dev/tty > /dev/tty 2> /dev/tty

```

18.3.5. Alternative *Chroot* environment

It has proven useful to provide separate *chroot* environments for the different Debian branches.

For this a copy of the directory `/var/cache/pbuilder/base.cow` is created. The file `/etc/apt/sources.lists` can then be adapted accordingly.

When updating packages, it may be necessary to update dependencies of these packages first. However, these are only available in the repo with a time delay. Here it can help for the further tests already once the line

```
deb http://incoming.debian.org/debian-builddd builddd-unstable main
```

in the `/etc/apt/sources.lists` *chroot*. This makes the packages there available for building in the *pbuilder* *chroot*.

18.4. More chroot systems

Besides building in an *pbuilder* chroot, there are other situations where using a separate system can be useful. This is especially true for running *mh_make*. (Chapter 44.3, page 364)

This setup of a *Maven chroot* is described as an example:

First, install the *debootstrap* package if it is not already present.

```
sudo mkdir --parents /srv/maven-chroot
```

ein entsprechendes Verzeichnis angelegt. Die Chroot selber wird mit

```
sudo /usr/sbin/debootstrap --arch amd64 sid \
/srv/maven-chroot http://ftp.de.debian.org/debian
```

.[32]

After that, the *root* user can start a new root directory with the *chroot* command.

Die konkreten Befehle lauten (als root):

```
# mount --options bind /proc /srv/maven-chroot/proc
# mount devpts /dev/pts --types devpts
# LANG=C chroot /srv/maven-chroot /usr/bin/bash
```

Die letzte Zeile startet die angelegte Chroot.

This user *root* can then no longer access files outside the new root directory.

The chroot environment can be removed again as follows:

```
sudo umount /srv/maven-chroot/proc # Unmount first!
sudo rm -rf /srv/maven-chroot/
```

18.5. Set up quilt for patching

With this script, *dquilt* is used for patching, among other things. This is a *debian* specific customization for *quilt*.

To create this customization, the file *.quiltrec-dpkg* with the following content ² is needed:

```
70 <DQuilt 70>≡
d=. ; while [ ! -d $d/debian -a 'readlink -ev $d' != / ]; do d=$d/..; done
if [ -d $d/debian ] &&[ -z $QUILT_PATCHES ]; then
    # if in Debian package 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
```

²<https://www.debian.org/doc/manuals/maint-guide/modify.html>

For manual operation, this also includes the entry in the `~/.bashrc` file. This entry looks like this:

```
# is needed for patch tracking with Quilt
alias dqilt="quilt --quiltrc=${HOME}/.quiltrc-dpkg"
complete -F -quilt-completion $_quilt_complete_opt dqilt
```

Note that settings in the `~/.bashrc` file are ignored when this program is executed. So all necessary settings must be mapped completely in the script.

The use of *quilt* or *dqilt* is described in chapter *Using Quilt* (chapter 33.2, page 263).

19. Set up Git

19.1. Branches

The Git repository of a Debian package usually has at least the following branches:

- `debian/sid`
- Upstream
- `pristine-tar`

The branch *debian/sid* can also be called *master* or *main*.

There can be an additional branch for *experimental*. Branches can also be created for *backports* and *update-proposal*.

Such further branches can also be created by the program script. (Chapter 41.1, page 355)

19.2. Mergen

If merged from *debian/experimental* to *debian/sid*, a fast-forward merge is usually performed.

This is the case when - as here - further updates were initially included in *debian/experimental* for testing, but then development is to be continued in *debian/sid*.

However, if changes have also been added to the *debian/sid* branch in the meantime, only a recursive merge can be performed.

If necessary, some customizations, e.g. in the *debian/* directory must be selected individually and added to the respective branch (*git cherry-pick*).

A command line proven for this purpose is:

```
git cherry-pick --edit -x <commit>
```

If multiple commits are to be fed to the branch at once, the following command line applies:

```
git cherry-pick --no-commit <commit> <commit> \dots
```

19.3. *gbp.conf*

The program script uses the applications from the Debian package *git-buildpackage*.

git-buildpackage (*gbp*) can and should be configured.

The configuration file *gbp.conf* is used to control this application. It can be placed at different locations in the file system.

19.3.1. Sequence

The configuration files for gbp are read in the following order:

1. /etc/git-buildpackage/gbp.conf, the system-wide configuration file
2. ~/.gbp.conf, the user specific configuration file
3. debian/gbp.conf, configuration for the repository or branch
4. .git/gbp.conf, configuration for the local repository

All configuration files have the same format.¹

By setting the environment variable `GBP_CONF_FILES` this order can be overridden. The content of this variable can be determined with `echo $GBP_CONF_FILES`.²

19.3.2. Sections in the *gbp.conf*

There are several sections in the *gbp.conf*. These sections are all optional.

For each *gbp* command³ a separate section can be created. In addition, there is a section that applies to all commands

Some important sections are listed below.

[DEFAULT] Options specified in this section are applied to all *gbp* commands.⁴

[import-orig] The options of this section overwrite those of the section *[DEFAULT]*. They are applied to the *gbp import-orig* command.

[pq] The options of this section are applied to the *gbp pq* command and override those of section *[DEFAULT]*.

[dch] The options of this section are applied to the *gbp dch* command and override the options of the section *[DEFAULT]*.

[buildpackage] The options of this section are applied to the *gbp buildpackage* command and override the options of section *[DEFAULT]*.

19.3.3. Syntax of the options

The options in the sections of *gbp.conf* are formed from the command line options. The possible options for the individual *gbp* commands can be taken from the respective man page.

These are specified without the introductory double minus sign. For example, `--patch-num-format=%02d` as a command line option becomes `patch-num-format=%02d`.

In the case of *gbp buildpackage*, *git-* must also be omitted⁵.

So the entry `pbuilder-options=PBUILDER_OPTION` in the *gbp.conf* corresponds to `--git-pbuilder-options=PBUILDER_OPTION`.

19.3.4. Example

In the user's home directory, a `~/.gbp.conf` file can be created as follows, for example:

74 `<gbp.conf 74>≡`

¹[3], section *Configuration Files* and the Manpage for *gbp-conf*

²[3] Section *Configuration Files/Overriding Parsing Order*

³[3] Manpage for *gbp-conf*

⁴[3] Manpage for *gbp-conf*

⁵Manpage for *gbp-conf*[3]

```
[DEFAULT]
sign-tags = True
# keyid for signing the package
keyid = 0x<keyid>
pristine-tar = True
# If you want to use normally sbuild
# builder = sbuild

[buildpackage]
postbuild = lintian $GBP_CHANGES_FILE
cleaner = /bin/true
# If you want to use normally pbuilder
# pbuilder = True
pbuilder-options = --source-only-changes --hookdir /home/mechtilde/.pbuilder

#[buildpackage]
# use a build area relative to the git repository
# export-dir=../build-area
# to use the same build area for all packages use an absolute path:
#export-dir=/home/debian-packages/build-area

[dch]
id-length = 7

# Options only affecting gbp pq
[pq]
#patch-numbers = False
# The format specifier for patch number prefixes
#patch-num-format = '
patch-num-format = '
# Whether to renumber patches when exporting patch queues
#renumber = False
renumber = True
# Whether to drop patch queue after export
#drop = False
```

The `/etc/git-buildpackage/gbp.conf` file lists the configuration options.

19.4. Git repositories on own infrastructure

19.4.1. Local Git repository

The local Git repository is either created by the script (chapter 29.4, page 136) or generated by cloning (chapter ??, page ??). Also, it can be created with *gbp import-dsc* (chapter 29.6, page 155).

More branches can be added to it (chapter 41.1, page 355).

19.4.2. Own Git server

The local Git repository can also be "mirrored" on its own Git server.

The setup of the server is done manually. The use of *cgkit* or *gitweb* facilitate access.

In the program script the name or IP of the own Git server can be entered (chapter 41.2, page 356).

The "workflow" is then as follows: After creating the configuration file, the name or IP of the own Git server is entered first (chapter 41.2, page 356) before starting to build a new package (chapter 29, page 109).

The "finished" package can then be uploaded to your own Git server (chapter 39.2, page 336).

20. *Salsa*-Repositories

Salsa is the name of a collaborative development server for **Debian**, based on the **GitLab** software. *Salsa* is intended to provide the necessary tools for collaborative development for package maintainers, packaging teams, and other Debian-related individuals and groups.[33]

Salsa provides all the features of **GitLab**. The service is available at <https://salsa.debian.org>. There is documentation of the debian-specific idiosyncrasies [34], which you should familiarize yourself with first.

20.1. *Salsa*-Konto anlegen

Creating an account on *Salsa* is very simple. One calls the page https://salsa.debian.org/users/sign_up, which is self-explanatory.

20.2. Creation of a *Salsa*-Repository

A **Git** repository on *salsa.debian.org* is not set up by the program script.

After login and authentication to <https://salsa.debian.org>, a new repository can be created there in the respective team.

For the creation of new projects on <https://salsa.debian.org> appropriate rights are required. These can be the rights of a **Debian Developer**. For team-supervised projects, these rights can also be assigned to other persons by the supervisors.

A description of the creation of a project in the Java team is given in chapter 20.3, page 78.

After logging in to the <https://salsa.debian.org> page and selecting the appropriate project for the new package, clicking on the *New Project* button calls the corresponding page. There you enter the project name. This is usually the name of the source code package. As visibility level *Public* is selected. Then the button *Create project* follows.

On the following web page there are some more explanations. Much of it is first created locally with the program described.

In the left navigation bar, in the *Settings* area, further configurations are now made for the project.

Here it is important to specify the path and the file name used under *CI/CD*.

CI stands for Continuous Integration

CD stands for Continuous Delivery

The file used here is called *salsa-ci.yml* (chapter 32.4.9, page 246) in the *debian/* directory.

The build script enters the salsa repository as "remote repository" and reminds the user to attach to salsa.debian.org (chapter 29.4.2, page 147)

20.3. Salsa-repository for the Java team

Salsa repositories that are assigned to a special project (such as the **Java** team) should be created as uniformly as possible. Often a script is provided by the project, which should be used for this purpose.

To do this, change to the desired team directory.



Figure 20.1.: Information about Java-Team^a

^aSource:<https://salsa.debian.org/java-team/>

20.3.1. Source of the Script

Under

<https://salsa.debian.org/java-team/pkg-java-scripts/blob/master/setup-salsa-repository> the script of the *Java team* can be downloaded. It is also attached in the appendix (chapter 47.1, page 387).

20.3.2. Dependencies

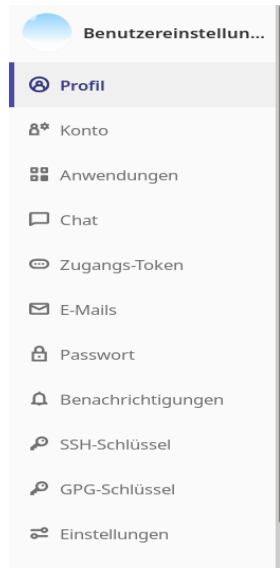
This *setup script* uses *jq*. *jq* is a lightweight and flexible *JSON* processor for the command line. It has minimal runtime dependencies. There is a **Debian** package of the same name. This package must be installed locally (chapter 18.1.2, page 58).

20.3.3. Get access token

A project specific token is still needed for the customization.

After logging in to *salsa.debian.org*, *Preferences* is selected from the user's dropdown menu. This will bring up the logged in user's *User Preferences* page.

In the left bar there is now the entry *access token*. There the page https://salsa.debian.org/profile/personal_access is called to create such a token. It is generated anew for each project.

Figure 20.2.: Create access token^a

^aSource:<https://salsa.debian.org>

There the name of the project to be created is entered. Furthermore, an expiration date for the token is entered. Afterwards the validity range of the token is specified. As *Scope api* is to be selected here. By clicking on the *Create personal access token* button, the access token appears at the top of the page.

20.3.4. Register token

The generated token is to be entered into the script as `SALSA_TOKEN`. The comment character is to be removed

It seems to make sense to store this script in the project directory in each case.

20.3.5. Call script

The script is now called with the name of the new project (package) as parameter:

```
./setup-salsa-repository.sh <packagename>
```

After that the following messages are output (using the BeanValidationApi example):

```
./setup-salsa-repository.sh beanvalidation-api
Creating the beanvalidation-api repository\dots
Configuring the BTS tag pending hook\dots
Configuring the KGB hook\dots
Configuring email notification on push\dots
```

Done! The repository is located at
<https://salsa.debian.org/java-team/beanvalidation-api>

April 6, 2025

If only the first line appears, check the script used (see chapter 47.1, page 387) and tokens.

20.4. Tasks on *salsa.debian.org*

20.4.1. Merge Request

Sometimes there are also so-called merge requests, in which others provide patches.

On *salsa.debian.org* there is then an entry "Merge-Requests" in the left navigation bar. There this can then be edited.

This is done by clicking the commit message.

21. Packaging beyond the branch *Unstable*

There are several reasons why it is allowed and encouraged to deviate from the general way of packaging new upstream versions exclusively for *Unstable* = *sid*. The program script also allows this (chapter 36, page 309).

The developer reference [9] refers to uploading to the *Stable* and *Oldstable* distributions as a "special case".¹

A major reason is the presence of a serious error or a security problem. Serious errors in a package are usually reported by an appropriately classified error report.

Another deviation from the general rule exists regarding the packages of the Mozilla suite, **Firefox** and **Thunderbird**. These packages will also be made available as *Security Updates* (chapter 21.1, page 82) for the *Stable* release promptly after their upstream release for the *ESR* = *Extended Support Release* version.

This can result in incompatibilities with the then current version in the *Stable* release, e.g. for the extensions, described here as a web extension (chapter 14, page 49). This is also a reason to update the *Stable* release (see also chapter 21.2, page 82).

In addition, it is often desirable to use more up-to-date versions of desktop applications in a stable operating system environment. For this purpose, the desired packages are made available as so-called *backports* (i.e. backports). (see chapter 21.3, page 84)

There are also good reasons to upload packages to *Experimental* (also called *rc-buggy*) first. This is especially true for new packages. During the period when existing versions are "frozen" and only bug fixes are allowed for the next release, new versions are often uploaded in advance to *Experimental*. (see chapter 21.5, page 84).

¹Chapter 5.5.1 in the Developer Reference[9]

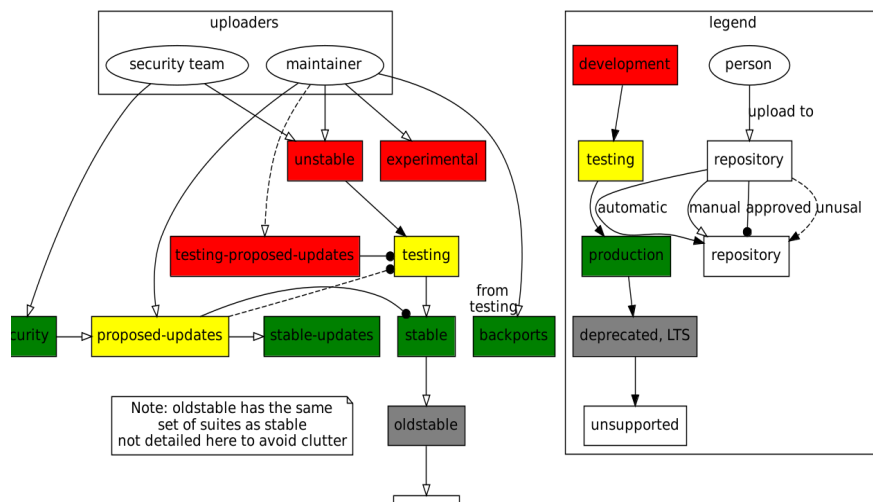


Figure 21.1.: Workflows [35] ^a

^a©2016 Antoine Beaupré anarcat@debian.org, CC-BY-SA 4.0

21.1. Security-Updates

It is important to backport changes to source code files because of security issues.

Whenever a security issue becomes known, the maintainer should work with the security team to provide a corrected version for the *Stable* or *Oldstable* release. The bug fix should be targeted. Further information can be found in the chapters 3.1.2 and 5.8.5 of the Debian Developer Reference[9].

21.2. (Old-)Stable-Proposal

If packages with serious bugs that do not affect IT security are to be updated, this can be done in *Proposed updates* under certain conditions. The same applies with regard to extensions for **Firefox** and **Thunderbird**.

The following have been mentioned as criteria for adding packages to *stable-updates* ²:

- The update is urgent and not security relevant. The security updates are done in the way mentioned above (chapter 21.1, page 82).
- The package in question is a data package, and the data must be updated in a timely manner (e.g., tzdata).
- Corrections to packages that are affected by external changes and on which no other or only a few other packages depend.
- Packages that need to be up to date to be useful (e.g. clamav).

Anyone who believes that the requested update meets these criteria should confidentially contact the release team via the debian-release@lists.debian.org mailing list, and explain that this update should also be done via *stable-updates*.

²<https://lists.debian.org/debian-devel-announce/2011/03/msg00010.html>

The package intended for backporting this way should have already been released in *Unstable* or better yet in *Testing*.

While backports are collected in a separate repository, *Proposed updates* are added to the *Stable* Proposed repository. These packages are uploaded to *Stable* by the Debian developers. This applies to oldstable-proposed updates accordingly. They will be published in the next point release.

21.2.1. Bug report

A necessary prerequisite is also here first a meaningful error report (bug report) (see also chapter 22, page 87).

This error report requires the following parameters:

```
Package: release.debian.org
Severity: normal
Tags: <ReleaseName>
User: release.debian.org@packages.debian.org
Usetags: pu
```

This bug report is generated against the *release.debian.org* pseudo package. The severity of such a bug report is generally at most *normal*. The subject will list the name of the package being built for *proposed-updates*.

Proposed updates are only allowed under special circumstances, as outlined. The *Release Team* will decide whether to release.

Therefore, the bug report must contain a reason why this version should be updated in *stable*. If there is already an associated bug number, it must be provided. This bug report **must** have a severity of *important* or higher.

21.2.2. Requirements for a patch

It should be possible to fix an error with a patch that is as small and accurate as possible. This is to prevent new errors from arising. Also, no new dependencies should be added. It must also be taken into account which other packages depend on this package.³

These matching patches are documented with *debdiff* *<PreviousPackage>.dsc* *<NewPackage>.dsc* *>* *<filename>.txt*. It specifies the difference that should be made between the current package version in stable/oldstable (previous package) and the version that should be uploaded (new package). How to do this with the program script is described in chapter 37.6.1 (page 319).

21.2.3. Dependencies for Mozilla packages

A sufficient justification for *Webextensions* may also be the exception for the packages of the Mozilla suite mentioned above (see chapter 21, page 81).

For these extensions namely incompatibilities with the then current version of **Firefox** or **Thunderbird** in the *Stable* release can arise. Then the previous versions are unusable, which is a considerable problem, especially in production use. (Chapter 36, page 309)

³Chapter 5.5.1 in the Developer Reference[9]

21.3. Stable-Backports

To be able to build packages for backports, some preparations have to be made.

Often such a package cannot be easily built in an *Stable* environment.

Then further packages from backports are needed. Packages from backports are not automatically installed in such a *chroot*.

Therefore it is necessary to enable such an installation in */etc/apt/preferences*.

For this purpose *hooks* are used in the *pbuilder* (chapter 18.3.3, page 65).

https://wiki.debianforum.de/Pbuilder_-_personal_package_builder

21.4. Backports-Repository

The packages for the unstable branch are built regularly, and from there - provided no errors are found - they are transferred to the testing branch. On release, the testing branch is first frozen and then becomes the new stable branch. A new testing branch is then opened.

If program versions from *testing* or (exceptionally ⁴) also from *unstable* are transferred to an earlier release, this is called *backporting*.

Due to the practice oriented release cycles of the stable Debian version, it is sometimes desirable to have software packages or newer versions from testing also available under stable.

The backports repository is used for this purpose.

Versioning (e.g. +deb9u1)

Backports are available for the stable and oldstable release.

21.5. Experimental

21.6. Backporting of unfamiliar packages

pristine-tar NMU Dependencies from Backports

Reference to (Chapter ??, Page ??) ⁵ for release after *Proposed updates* for *Stable* or *Old-Stable*.

21.7. Versioning

The cause or origin of the upgrade also affects the versioning. It must be ensured that *dpkg* can correctly interpret newer versions as upgrades. A look at the next expected version number can help..

When building for *experimental*, the following version entry is recommended in the file *debian/changelog*:

<version number>-<revision number>~exp<sequential number>

The revision number is usually: 1

When building for *Proposed updates*, there are two cases to distinguish..

⁴usually only security updates

⁵[9] Section *upload-stable*

Case A A version existing in the *Stable* release is to be corrected while retaining this version number.

Case B Exceptionally, a new version is to be included in the release.

For the version entry in *debian/changelog* it is mandatory to use the following nomenclature:

Case A <original version and revision number> +deb<Debian release number>u<revision number of the update>

Case B <version and revision number from unstable/testing>~deb<Debian release number>u<revision number of update>.

Using the ~causes the version in *Stable* to be smaller than the one in *Testing* (for the next release). This secures the update path.

22. An email for the start

Before packaging with the goal of publishing in the Debian repository can begin, an email is required. This triggers a "bugreport", whose number is to be noted in the file *debian/changelog*. By the publication of the package this bug report is to be closed namely then.

For more information, visit <https://www.debian.org/devel/wnpp/>

22.1. ITP - Intent To Package

ITP means "Intent to Package". This means that this package is being worked on.

This is the name of a bug report created against the *WNPP* package, which means *Work-Needing and Prospective Package*. This can be translated as *Work-Needing and Prospective Package*.

This should be announced in good time. Maybe there are still hints what to consider with the planned package. Also, this prevents different groups from trying to package this package for Debian.

One possibility is to do this with the `reportbug` tool installed on every Debian system. But there is also the possibility to simply use an email template. The following template is based on the *itp_template* with which *reportbug* creates this email.

```
To: submit@bugs.debian.org
Subject: ITP: <Source Name> - <Short Description>
Package: wnpp
Severity: wishlist

* Package name : <Package Name>
Version : x.y.z
Upstream Author : Name <somebody@example.org>

* URL : http://www.example.org/
* License : (GPL, LGPL, BSD, MIT/X, etc.)
Programming Lang: (C, C++, C#, Perl, Python, etc.)
Description : <Short Description>

(Include the long description here.)

<And answer following questions:>

* Why is this package useful/relevant?
* Is it a dependency for another package?
```

April 6, 2025

- * Do you use it yourself?
- * If there are other packages providing similar functionality, how does it compare?
- * How do you plan to maintain it? Do you plan to maintain it inside a packaging team? (check .)
- * Are you looking for co-maintainers? Do you need a sponsor?

The text in the first line after the *To:* goes in the address line. The text in the second line after the *Subject:* goes in the subject line, with the placeholders replaced by the name of the source code and a short description.

If the package is uploaded, this number from the bug tracking system must be noted in the *debian/changelog* (Closes:#XXXXXXXXX)

22.2. RFP - Request For Package

RFP means *Request for Package*. This means that such a package is desired in Debian.

I once created an email template for this as well.

To: submit@bugs.debian.org
Subject: RFP: <Sourcecode Name> - <Short Description>
Package: wnpp
Severity: wishlist

* Package name : <Package Name>
Version : x.y.z
Upstream Author : Name <somebody@example.org>

* URL : http://www.example.org/
* License : (GPL, LGPL, BSD, MIT/X, etc.)
Programming Lang: (C, C++, C#, Perl, Python, etc.)
Description : <Short Description>

(Include the long description here.)

<And answer following questions:>

- * Why is this package useful/relevant? Is it a dependency for another package?
- * Do you use it yourself?
- * If there are other packages providing similar functionality, how does it compare?
- * How do you plan to maintain it? Do you plan to maintain it inside a packaging team? (check .)
- * Are you looking for co-maintainers? Do you need a sponsor?

Again, the text in the first line after the *To:* goes in the address line. The text in the second line after the *Subject:* goes in the subject line, with the placeholders replaced by the name of the source code and a short description.

22.3. ITA - Intent To Adoption

This is used when a package that is marked with "O" or "RFA" is to be taken over.

For this purpose, the previous error report must be renamed and "O" or "RFA" must be replaced by "ITA".

In doing so, you enter yourself as the owner.

If a bug report is to be renamed or the owner changed, this must be done by email to *control@bugs.debian.org* or directly to the bug report via the number (xxxxxx@bugs.debian.org)¹.

A structured *pseudo header* must be used.²

22.4. RFA - Request for Adoption

22.5. RFH - Request For Help

22.6. O - Orphaned

22.7. RFS - Request For Sponsor

As described on <https://mentors.debian.net/sponsor/rfs-howto>, the template was adjusted.

```
To: submit@bugs.debian.org
Subject:
ITP: <Package Name> - <Short Description>
Package: sponsorship-request
Severity: normal [important for RC bugs, wishlist for new packages]
```

Dear mentors,

I am looking for a sponsor for my package "<Source Name>":

```
* Package name : <Source Name> Version : x.y.z
Upstream Author : Name <somebody@example.org>
* URL : http://www.example.org/
* License : (GPL, LGPL, BSD, MIT/X, etc.)
Programming Lang: (C, C++, C#, Perl, Python, etc.)
Description : <Short Description>
```

It builds those binary packages:

<Name of the Binaries>

To access further information about this package, please visit the following URL:

¹<https://www.debian.org/devel/wnpp/>

²<https://www.debian.org/Bugs/Reporting#control>

April 6, 2025

`https://mentors.debian.net/package/<package name>`

Alternatively, one can download the package with `dget` using this command:

`dget -x https://mentors.debian.net/debian/pool/main/<p>/<package name>/<package name>_x.y.z.d`

Changes since the last upload: [your most recent changelog entry]

Regards,

22.8. Changes to the bug report

In the course of such a process it may happen that changes have to be made to the bug report. Such a change can be a change of the maintainer, the title or something else.

For this purpose the commands of the *control email server* are used. The description of this can be found at <https://www.debian.org/Bugs/server-control>

This can also be done with a defined structured email. It will be sent to *control@bugs.debian.org*. The subject should be

For the change of the title, an additional line is then added as follows.

`Control: retitle -1 <neuer Titel>`

added.

The following line is added for the maintainer change:

`Control: owner -1 <Neuer Maintainer or wnpp@debian.org>`

If necessary, an erroneously closed report must be reopened. This is done with

`Control: reopen bugnumber [address of the author | = | !]`

22.9. *usertags* added

In the course of a maintainer's life it happens again and again that error messages must or should be tagged.

For example, it is helpful for so-called *bug squashing parties* to also mark the planned and executed bug fixes.

For this, an email is written to the bug report. Address is then *<bugnumber>debian.org*. At the same time this mail should also go *CC* to *controlbugs.debian.org*.

At the beginning of such an e-mail is then:

```
user debian-release@lists.debian.org
usertags -1 + <Title of the BSP>
thank you
```

23. Set up report bug

The Command-Line interface is already part of the basic installation. In addition, a graphical user interface can be installed with the package *reportbug-gtk*.

24. Autopkgtest

25. Reproducible builds

25.1. Configuration of *sbuild*

After installing the required packages:

```
sudo apt install sbuild schroot debootstrap apt-cacher-ng devscripts
```

the configuration file *sbuild* is created[SBUILD2022].

This is done by copying the following section to the `~/.sbuilddrc` file. These settings make it possible to avoid long command line options for typical workflows and run all tests after build.

```
95a <.sbuild.rc 95a>≡
#####
# PACKAGE BUILD RELATED (additionally produce _source.changes)
#####
# -d
$distribution = 'unstable';
# -A
$build_arch_all = 1;
# -s
$build_source = 1;
# --source-only-changes (applicable for dput. irrelevant for dgit push-source).
$source_only_changes = 1;
# -v
$verbose = 1;
<.sbuild.rc5 95b>
```

Adapt parallel=5 to the resources of your system.

```
95b <.sbuild.rc5 95b>≡ (95a)
# parallel build
$ENV{'DEB_BUILD_OPTIONS'} = 'parallel=5';
<.sbuild.rc6 95c>
```

If instead the variable `$run_lintian` is set to 0, the execution of lintian is disabled.

Any post-build package test function can be turned off by setting the corresponding variable to 0.

```
95c <.sbuild.rc6 95c>≡ (95b)
#####
# POST-BUILD RELATED (turn off functionality by setting variables to 0)
#####
$run_lintian = 1;
$lintian_opts = ['-i', '-I'];
<.sbuild.rc7 96a>
```

If the variable `$run_piuparts` is set to 0, the execution of `piuparts` is disabled.

```
96a <.sbuid.rc7 96a>≡ (95c)
    $run_piuparts = 1;
    $piuparts_opts = ['--schroot', 'unstable-amd64-sbuid'];
    <.sbuid.rc8 96b>
```

The `'--no-eatmydata'` option for `piuparts` is needed when configuring `schroot` with `"command-prefix=eatmydata"` in `/etc/schroot/chroot.d/unstable-amd64-sbuid-*`. Then it must be added at the end – after a comma.

If you set the `$run_autopkgtest` variable to 0 instead, `autopkgtest` execution is disabled.

```
96b <.sbuid.rc8 96b>≡ (96a)
    $run_autopkgtest = 1;
    $autopkgtest_root_args = '';
    $autopkgtest_opts = [ '--', 'schroot', '%r-%a-sbuid' ];

    %r-%a-sbuid' ];
    ##### # PERL MAGIC #####
```

For more options, see the `sbuid.conf` man page.

Besides this, the user must be set up with

```
sudo sbuid-adduser \${LOGNAME}
```

in the `sbuid` group. This will add your username so that it can use the `sbuid` command.

The following message appears after the user is set up:

Now try a build:

```
cd /path/to/source sbuid-update -ud <distribution> (or "sbuid-apt <distribution> apt-get -f
```

This means that `sbuid-adduser` copies the `sbuid` template configuration in `/usr/share/doc/sbuid/examples/example.sbuilddrc` to each user's `~/.sbuiddrc` to use as the `sbuid` configuration for that user. The `sbuid` settings can be customized here. Usually, however, no adjustments are necessary. If they are, they should be done once per user.

The creation of the `Sbuid` chroot is described in chapter 34.5.1 (page 295).

25.2. reprotest

26. piuparts

27. Overcome difficulties

27.1. Unfreeze a package

One difficulty arises from the fact that there is a period of time before a scheduled release when packages no longer automatically migrate from *unstable* to *testing*.

In the full ‘freeze’, all packages that are yet to migrate from *unstable* to *testing* require an unblock by the release team.[36]. This must be requested with an *unblock bugreport*.

27.1.1. Request for unblocking

To do this, first create a file with *debdiff* (see also chapter 21.2, page 82).

This creates the difference between the version in *testing* (old version) and *Unstable* (new version) with the respective *dsc*.

This difference file should be checked to make sure that it does not have any unimportant changes for the desired fix.

Then, using the *reportbug* tool, a bug report is generated against the *release.debian.org* package and the difference file is attached. This bug report contains a detailed justification for the changes and references to bug numbers. Likewise, it contains a concise description of the problem that was fixed.

The following questions should be addressed.

```
Package: release.debian.org
User: release.debian.org@packages.debian.org
Usertags: unblock
Severity: normal
```

Please unblock package <source name>

(Please provide enough (but not too much) information to help the release team to judge the request.)

[Reason] (Explain what the reason for the unblock request is.)

[Impact] (What is the impact for the user if the unblock isn't granted?)

[Tests] (What automated or manual tests cover the affected code?)

[Risks] (Discussion of the risks involved. E.g. code is trivial or complex, key package vs 1

[Checklist] [] all changes are documented in the d/changelog [] I reviewed all changes and

[Other info] (Anything else the release team should know.)

27.2. Fix release critical bugs

Before a new release, it is often necessary to help maintainers fix bugs that prevent the package from being released.

This often happens at events organized for this purpose ¹.

Under <https://www.debian.org/doc/manuals/developers-reference/pkgs.html#non-maintainer-uploads-nmus> the desired procedure is described.

It is essential that in the file *debian/changelog* (chapter 34.1, page 277 a corresponding entry is made in the second line.

* Non-maintainer upload

27.3. Remove package from repository

The developer reference ² also describes how to remove a package.

To do this, it must first be determined that no other package requires this as a dependency.

27.3.1.

A bug report must now be generated for the execution.

¹<https://wiki.debian.org/BSP>

²[9], Section 5.9 Remove packages

Part III.

How a shell script helps to build a Debian package

28. First steps in the program script

Now we start with the program script. This helps to build a **Debian** package and supports the upload of the same. Thereby the program flow gives a purposeful sequence of the necessary steps. The program script builds thus no **Debian** packages, but supports as an assistant the Maintainer. This is also explicitly pointed out in the initial dialog. (Chapter 28.2, page 104)

The prerequisite is that all required packages are installed and the system is set up accordingly (chapter 18, page 57).

The program script has a modular structure, so that one can "get out" at many points and "get in" again later. So you don't always have to carry out the building process "in one go" until completion.

The program flow visible to the user starts in any case with a start dialog (chapter 28.2, page 104).

Building a new **Debian** package first requires creating a new project (chapter 29, page 109).

All the following points are at least essentially taken care of by the program script.

Configuration file The program requires a configuration file (chapter 29.1, page 109), which is initially created by the program script. This can be changed at any time.

System setup To build a **Debian** package, the following preparatory tasks, among others, must be performed:

Providing the required directories Creating the directories is described in chapter 29.2.2, page 131.

Setting up a Git repository Setting up the local Git repository (chapter 29.4, page 136) must be done before running *gbp import-orig* (chapter 31.4.10, page 218)

Providing the source code This is described in chapter 31.3, page 182.

Providing the required files in the *debian/* directory Creating the files in the *debian/* directory is done as part of building the **Debian** revision (chapter 32, page 225).

Build This is described in Chapter 34, page 277

Test – as much as possible This is described in Chapter 37, page 311.

Upload This is described in chapter 40, page 337.

It's a long way to uploading. But as we all know, even the longest way begins with the first step. It should still be noted that the program script is operated keyboard-driven (with the TAB key). Control with the mouse can lead to unexpected effects.

28.1. The beginning is at the end

The program script contains many functions.

The main program simply calls the *BuildApp* function. It is at the end of the program script.

```
104a  ⟨MainProgram 104a⟩≡ (157)
      #####
      # Here it starts
      BuildApp

      #####
      # This is the end, my friend
```

28.2. And this is what the user sees first.

The *BuildApp* function essentially controls the program flow by calling other functions.
The first thing it does is present the program to the user.

```
104b  ⟨BuildApp 104b⟩≡ (107)
      function BuildApp{
          # Called by main program

          #####

          # Intro

          #####

          intro="Assistent to build simple Debian packages\n
          using git-buildpackage\n
          Authors: Mechtilde Stehmann\n
                   Michael Stehmann\n
          Version: 0.8.4\n
          License: GPL v3+\n
          This program does not build Debian packages itself.
          It is only an assistant for the package maintainer."

          whiptail --title "Introduction" --msgbox "$intro" 20 60

      }
      ⟨BuildApp3 105⟩
```

The following welcome dialog appears for this purpose..

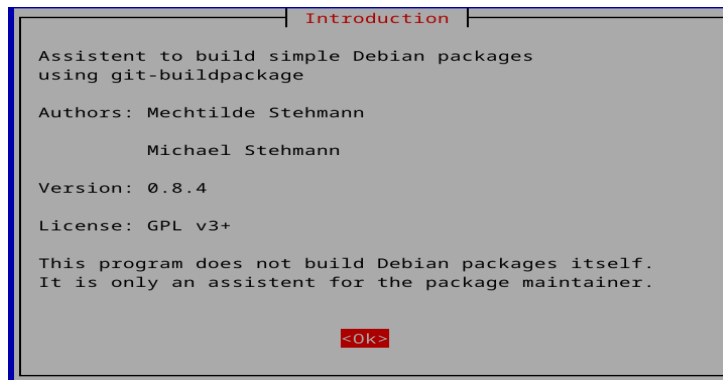


Figure 28.1.: Start screen

Then this function queries the name of the corresponding project with the *AskOrigName* function (chapter 28.3, page 106).

```

105  <BuildApp3 105>≡
      # Definitions of Project
      AskOrigName
      CreateDirsAndLogFile

      # Flag for additional gbp buildpackage options
      OptFlag=0

      #####

      # End of intro

<BuildApp7 132a>

```

(104b)

Finally, this function *BuildApp* checks the existence of the necessary local infrastructure (configuration file (chapter 29.1, page 109), directories (chapter 29.2.2, page 131), Git repository (chapter 30.4, page 163)) and takes care of their creation if necessary.

28.3. Request project name

The project name of an existing project is now queried or specified for a new project. The insertion of the project name can also be done by *Copy & Paste*. This possibility is a special feature of *whiptail*.

106a $\langle AskOrigName\ 106a \rangle \equiv$

```
function AskOrigName {
    # Called by BuildApp ConfigFileLEC and itself

    # Name of the project (without this name the app cannot work)
    OrigName=$(whiptail --title "This name is required!" \
        --inputbox "Name of the project:" \
        --cancel-button "Exit" 15 60 3>&2 2>&1 1>&3)
 $\langle AskOrigName1\ 106b \rangle$ 
```

(171)

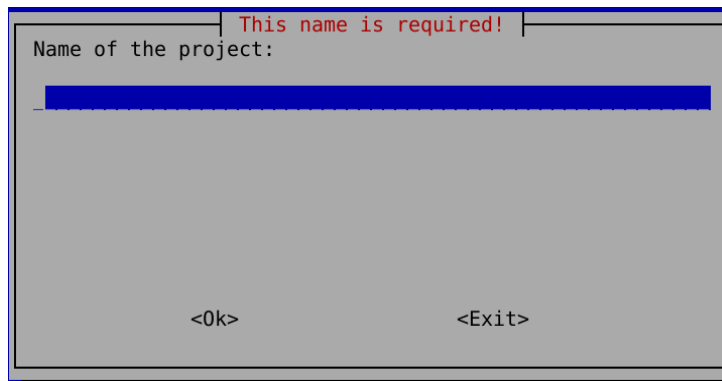


Figure 28.2.: Specification of the project name.

A *whiptail --inputbox* requires redirections of the output using file identifiers (*3>&2 2>&1 1>&3*).

106b $\langle AskOrigName1\ 106b \rangle \equiv$

```
if [ $? -ne 0 ]
then whiptail --title "Bye" --msgbox "Bye" 15 60
    exit
fi
```

(106a)

$\langle AskOrigName2\ 107 \rangle$

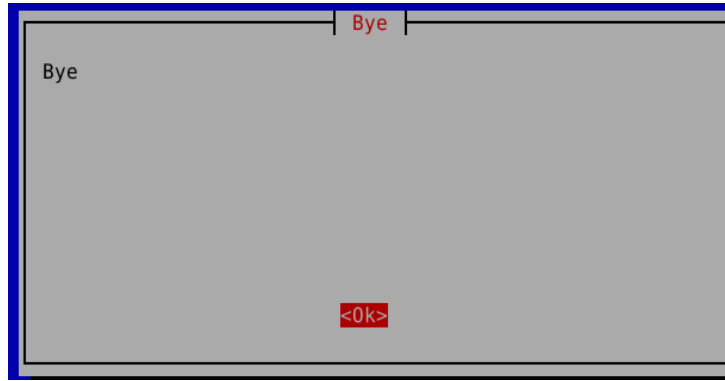


Figure 28.3.: Bye

Only the first three file identifiers (starting with 0) have a standardised meaning:

- 0 - stdin - standard input
- 1 - stdout - standard output
- 2 - stderr - standard error display

If the project already exists, it continues with the display of the configuration file (chapter 30.1, page 159), the selection of a Git branch (chapter 30.4, page 163) and then the task selection (chapter 30.5, page 171).

If **no** project name is entered, a note is displayed. The entry of a project name is mandatory. With *Exit* the programme is terminated.

```

107  <AskOrigName2 107>≡                                     (106b)
      if [ -z "${OrigName}" ]
      then whiptail --title "No project name" \
        --msgbox "You have to identify a project name\n \
        even it is a new project!" 15 60
        AskOrigName
      fi
      ConfigFileLEC
    }

<BuildApp 104b>

```

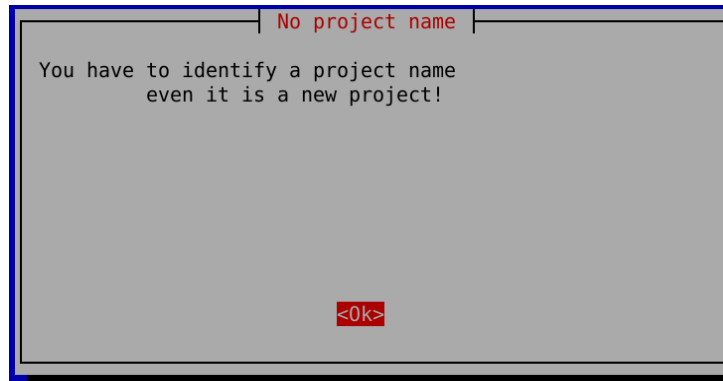


Figure 28.4.: No project name specified.

After confirming *OK*, the input dialogue is displayed again.

28.4. Next steps

The next steps now depend on whether a configuration file already exists for the project, as described in chapter 18.2.2.1, page 59.

In this case it is “loaded” (chapter 30.1, page 159).

Otherwise, the configuration file is created again.

29. Create a new project

To create a new project first the configuration file and the necessary infrastructure are created.

29.1. Create configuration file

The configuration file is stored in the user's home directory in the *.debian_project/* directory as the *<project name>* file.

```
109a  <ConfigFileLEC 109a>≡ (129)
      function ConfigFileLEC {
          # Called by AskOrigName CreateNewBranch TaskSelect OwnServer

          ## Load, edit or create config file - using AskConfig

          # Path to config files directory
          ConfigPath=~/.debian_project/
          changeflag=0

      <ConfigFileLEC1 159>
```

The function *ConfigFileLEC* checks first whether a configuration file with the project name exists. If the result of the check is negative, the message occurs that no configuration file with this name could be found.

```
109b  <ConfigFileLEC4 109b>≡ (161)
      else
          if whiptail --title "Config file not found" \
          --yesno "There is no config file for ${OrigName}\n \
          which you can edit.\n \
          Do you want to create a new project?" \
          --yes-button "Yes" --no-button "No" 15 60
          then
              changeflag=1 <ConfigFileLEC5 110>
```

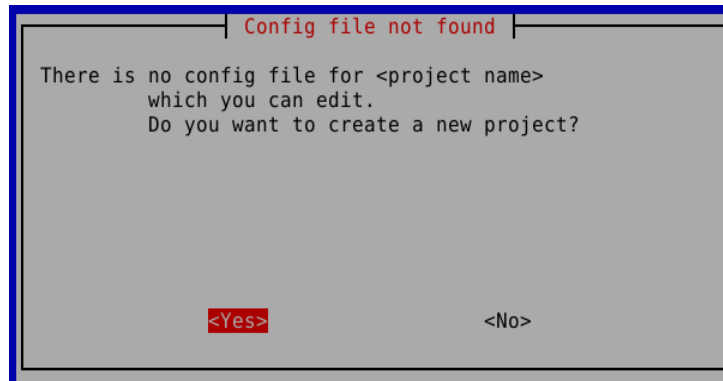


Figure 29.1.: No configuration file found.

A *whiptail -yesno* returns 0 or true if the question is affirmative, and 1 or false if it is negative.

If this question is answered in the affirmative, the directory *.debian_project* is also created as a precautionary measure as *ConfigPath*, if it does not already exist. Then the function *AskConfig* is called, which creates the configuration file.

```

110 <ConfigFileLEC5 110>≡ (109b)
      mkdir --parents ${ConfigPath}
      AskConfig
    else
      AskOrigName
    fi
  fi
<ConfigFileLEC6 128b>

```

The query whether to create a new project allows the user to correct typos when entering the project name, if he/she negates them.

The following sections discuss the variables included in the configuration file. First follow the variables that are required for all packages. Then follow the variables. which are only required for one group of packages at a time (for **Java** packages see chapter 29.1.2.2, page 121, for *webezt* packages see chapter 29.1.2.3, page 124).

29.1.1. Query common variables for configuration file

The function *AskConfig* assigns values to the variables to be contained in the configuration file in its first part. This is done by querying the individual variables. Saving the configuration file is then described in chapter 29.1.3, page 127.

It is checked whether a value has been assigned to the respective variable.

This is done first for the name of the source package, which is assigned as the value of the *SourceName* variable.

The name of the source package is the name that the upstream project has given to its software.

```

111 <AskConfig 111>≡ (121a)
    function AskConfig {
        # Called by ConfigFileLEC

        if [ -z "${SourceName}" ]
        then
            SourceName=$(whiptail --title "Source Package Name" \
                --inputbox "Name of the source package:" \
                --cancel-button "Exit" 15 60 3>&2 2>&1 1>&3)

            if [ $? -ne 0 ] # Cancel-Button was pressed
            then
                exit
            else
                changeflag=1
            fi
        fi
    }

```

<AskConfig1 112>

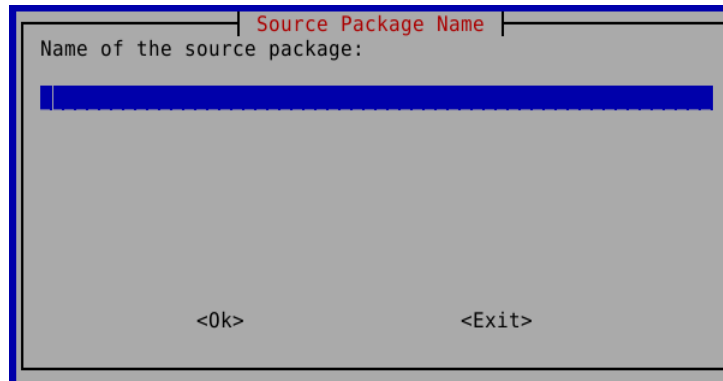


Figure 29.2.: Name of the source package

If the variable contains a value, it is asked whether this is the correct value. This serves on the one hand the control of the previous input and on the other hand this opens the possibility of editing (chapter 30.1, page 159) the configuration file by means of the function *AskConfig*.

```
112 <AskConfig1 112>≡ (111)
    if ! whiptail --title "Source Package Name" \
    --yesno "The name of the source package is ${SourceName}" \
    --yes-button "Yes" --no-button "No" 15 60

    <AskConfig2 113>
```

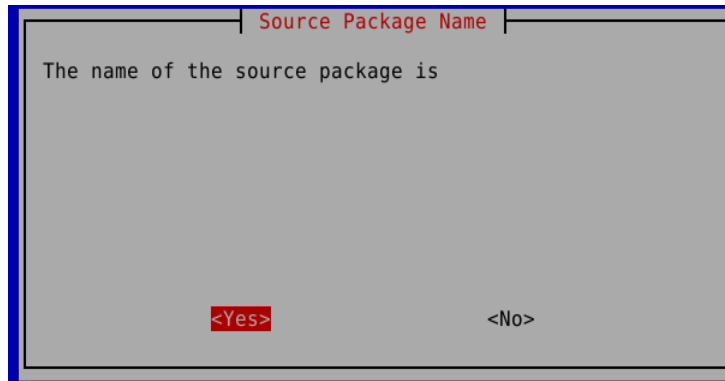


Figure 29.3.: Specify the name of the source package

```

113  <AskConfig2 113>≡ (112)
      then
        SourceName=$(whiptail --title "Name of the source package" \
          --inputbox "Real name of the source package:" \
          --cancel-button "Exit" 15 60 3>&2 2>&1 1>&3)

        if [ ${#SourceName} -eq 0 -o $? -ne 0 ]
        then
          exit
        else
          changeflag=1
        fi
      fi

  <AskConfig3 114>

```

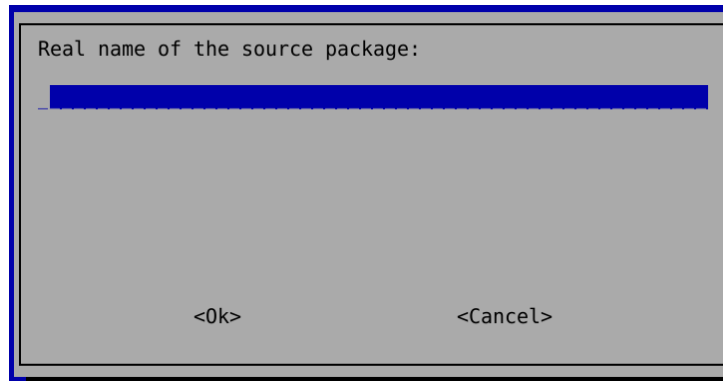


Figure 29.4.: Specify the correct name of the source package

Now the name of the package to be built is determined. For this first the value of the variable *SourceName* is assigned to the variable *PackName* as default value. Also for this a confirmation is queried and a correction possibility is opened.

```

114  <AskConfig3 114>≡
      if [ -z "${PackName}" ]
      then
          PackName=${SourceName} tadd=", too?"
      else
          tadd="?"
      fi

      if ! whiptail --title "PackName" \
          --yesno "The name of the package is ${PackName}${tadd}" \
          --yes-button "Yes" --no-button "No" 15 60
      <AskConfig3-1 115>
    
```

(113)

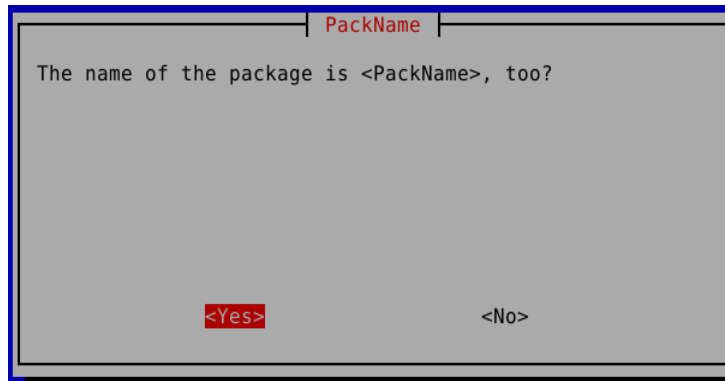


Figure 29.5.: Correct name of the package specified.

```

115  <AskConfig3-1 115>≡ (114)
      then
        PackName=$(whiptail --title "Name of the Debian Package" \
          --inputbox "Real name of the package,\nwhich should be built:" \
          --cancel-button "Exit" 15 60 3>&2 2>&1 1>&3)

        if [ ${#PackName} -eq 0 -o $? -ne 0 ]
        then
          exit
        else
          changeflag=1
        fi
      fi

  <AskConfig4 116>

```

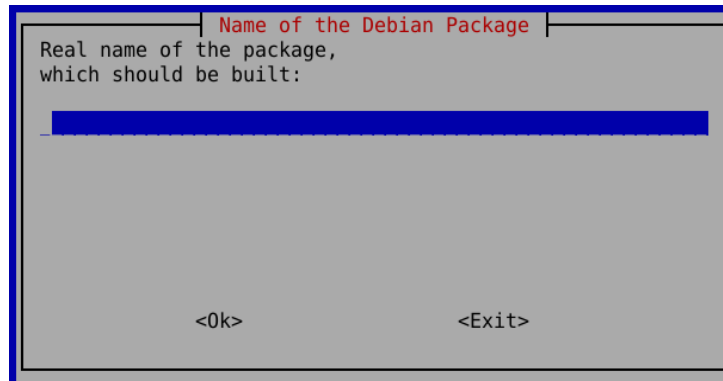


Figure 29.6.: Correct name of the package specified.

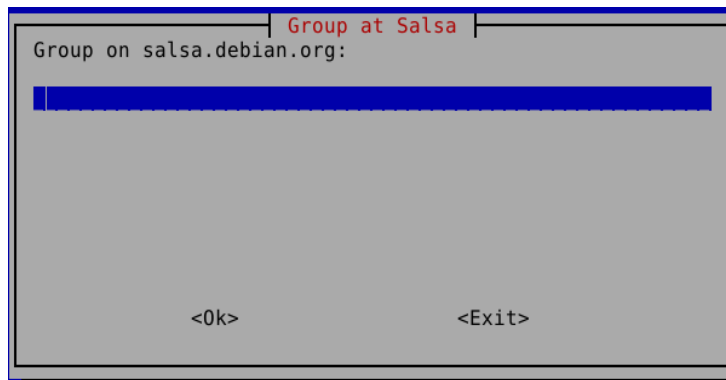
Now the name of the group is requested under which the Git repository on *salsa.debian.org* should be created.

```

116  <AskConfig4 116>≡
      if [ -z "${SalsaName}" ]
      then
        SalsaName=$(whiptail --title "Group at Salsa" \
          --inputbox "Group on salsa.debian.org:" \
          --cancel-button "Exit" 15 60 3>&2 2>&1 1>&3)
        if [ $? -ne 0 ]
        # Cancel-Button was pressed
        then
          exit
        else
          SalsaName="${SalsaName}/${SourceName}.git"
        fi
      fi
    <AskConfig5 117a>

```

(115)

Figure 29.7.: Name of the group specified on *salsa.debian.org*.

There are different packaging teams (for example for **Java** or **Python** packages)¹.

For the **python** team, the *packages* directory must also be specified, i.e. *python-team/packages*.

These have their own groups on *salsa.debian.org*. If a package is to be maintained independently of a packaging team, enter *Debian* as the group.

```
117a <AskConfig5 117a>≡ (116)
      if ! whiptail --title "Salsa Name" \
        --yesno "Group and project name of the repo on salsa.debian.org is $SalsaName" \
        --yes-button "Yes" --no-button "No" 15 60
      <AskConfig5-1 117b>
```

Figure 29.8.: Name of the group specified on *salsa.debian.org*.

```
117b <AskConfig5-1 117b>≡ (117a)
      then
        SalsaName=$(whiptail --title "Salsa Group and Project Name" \
          --inputbox "Real group and project name of the repo on salsa.debian.org:" \
          --cancel-button "Exit" 15 60 3>&2 2>&1 1>&3)

      <AskConfig5-2 118a>
```

¹https://wiki.debian.org/Teams#Packaging_teams

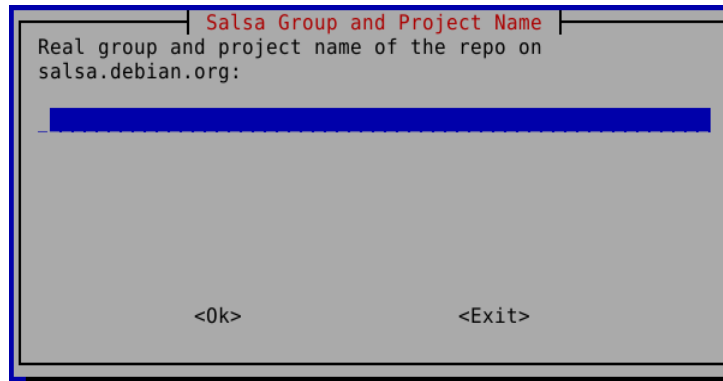


Figure 29.9.: Name of the group specified on *salsa.debian.org*.

```
118a  <AskConfig5-2 118a>≡ (117b)
      if [ ${#SalsaName} -eq 0 -o $? -ne 0 ]
      then
        exit
      else
        changeflag=1
      fi
    fi
```

<AskConfig6 118b>

With <https://salsa.debian.org/explore/groups> all public groups can be displayed.

It is now checked whether a file *DefaultValues* exists in the directory *~/.debian_project*. In this file, variables are assigned values that apply to many projects (chapter 18.2.2.2, page 60). The *DefaultValues* script is then executed.

```
118b  <AskConfig6 118b>≡ (118a)
      if [ -f ${ConfigPath}/DefaultValues ]
      then
        . ${ConfigPath}/DefaultValues
      fi
```

<AskConfig7 119>

The variable *DefaultProjectPath* is assigned as value the path which leads to the directory containing the individual project directories as subdirectories.

```

119  <AskConfig7 119>≡ (118b)
      if [ -n "${DefaultProjectPath}" ]
      then
          ProjectPath=${DefaultProjectPath}
      fi

      if [ -z "${ProjectPath}" ]
      then
          ProjectPath=$(whiptail --title "Path to Project Directory" \
            --inputbox "Path to the project directory on your local machine\n \
            (without '${OrigName}':" \
            --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
          if [ -z "${ProjectPath}" ]
          then
              echo -e "Path to the project directory on your local machine\n \
                (without '${OrigName}':"
              read ProjectPath
          fi
          changeflag=1
      fi

      if ! whiptail --title "ProjectPath" \
        --yesno "Path to the project directory on your local machine \
        is ${ProjectPath}/${OrigName}" \
        --yes-button "Yes" --no-button "No" 15 60
      then
          ProjectPath=$(whiptail --title "Path to Project Directory" \
            --inputbox "Real path to the project directory on your local machine\n \
            (without '${OrigName}':" \
            --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
          if [ -z "${ProjectPath}" ]
          then
              echo -e "Path to the project directory on your local machine\n \
                (without '${OrigName}':"
              read ProjectPath
          fi
          if [ -z "${ProjectPath}" ]
          then
              exit
          else
              changeflag=1
          fi
      fi
  fi

```

<AskConfig10 121b>

29.1.2. Query special variables for the configuration file

Now the variables are determined that are only needed for special types of packages. Starting with the variables for Java packages (chapter 29.1.2.2, page 121

In addition, the path to a required plugin is entered in the configuration file, if necessary.

29.1.2.1. Identifying the plugin paths

Certain types of packages are built using plugins to the (main) script. These are packages built with *maven*, *Mozilla* extensions, and *Python* programs.

The *DetectPlugins* function detects the paths to the plugin script files. It can also be used for other plugins.

When calling *DetectPlugins*, two options must be passed. The first option (*\$1*) is passed to *PluginName* and the second (*\$2*) to *PluginFile*.

```
120 <DetectPlugins 120>≡ (162a)
function DetectPlugins {
    # Called by AskConfig

    # This function needs two options: the name of the plugin
    # and the name of the plugin script file
    PluginName=$1
    PluginFile=$2

    # Determine path to the (needed) plugin script
    PluginPathL=$(locate ${PluginFile})
    PluginPathA=(${PluginPathL})

    # If there is only one result
    if [ ${#PluginPathA[@]} -eq 1 ]
    then
        PluginPath=${PluginPathA[0]}
    # If there are some results
    elif [ ${#PluginPathA[@]} -gt 1 ]
    then
        i=0; slct=''
        for element in ${PluginPathA[*]}
        do
            slct=$slct' '$i' '${element}' off '
            i=$(expr $i + 1)
        done

        PluginNr=$(whiptail --title "${PluginName} plugin found" \
        --radiolist "Select:" 15 60 5 $slct \
        --cancel-button "Cancel" 3>&2 2>&1 1>&3)

    <DetectPlugins3 121a>
```

```

121a  <DetectPlugins3 121a>≡ (120)
        if [ $? -eq 1 ]
        then
            return 24
        fi

        PluginPath=${PluginPathA[${PluginNr}]}
        # If there is no result
        else
            PluginFlag=0
            whiptail --title "File not found" \
                --msgbox "'${PluginFile}' was not located!" \
                15 60
            return 24
        fi
    }

    <AskConfig 111>

```

29.1.2.2. Variable query for Java packages

Certain types of packages require special handling. This is done, among other things, by separate scripts. This handling is controlled by "flags". This also avoids unnecessary loading of the plugin scripts.

```

121b  <AskConfig10 121b>≡ (119)
        # Java Flag
        # This is needed to trigger special entries
        # in debian/control and debian/rules
        if [ -z "${JavaFlag}" ]
        then
            if whiptail --title "Java" --defaultno \
                --yesno "Do you want to build a Java package?" \
                --yes-button "Yes" --no-button "No" --defaultno 15 60
            <AskConfig10-1 122>

```

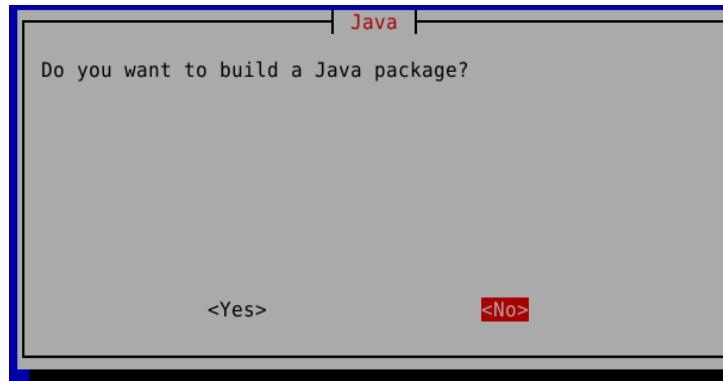


Figure 29.10.: Should a Java package be built?

```
122  <AskConfig10-1 122>≡ (121b)
      then
        JavaFlag=1
      else
        JavaFlag=0
      fi

      if [ ${JavaFlag} -eq 1 ] &&[ -z ${JavaPluginPath} ]
      then
        DetectPlugins 'Java' build-gbp-java-plugin.sh
        if [ $? -eq 0 ]
        then
          JavaPluginPath=${PluginPath}
        else
          # If 'DetectPlugins' returns 24
          JavaFlag=0
        fi
      fi
      changeflag=1
    fi

    <AskConfig11 123a>
```

It is also queried whether the Maven plugin (chapter 44, page 363) should be used. This is then localized in the system and the path to the script is stored in the configuration file.

```
123a  <AskConfig11 123a>≡ (122)
      # Maven Plugin
      if [ ${JavaFlag} -eq 1 ]
      then
        if [ -z "${MavenPluginFlag}" ]
        then
          if whiptail --title "Maven" --defaultno \
            --yesno "Should the (Java) package be built with maven?" \
            --yes-button "Yes" --no-button "No" --defaultno 15 60
          then
            MavenPluginFlag=1
          else
            MavenPluginFlag=0
          fi
          changeflag=1
        fi
      fi
```

<AskConfig12 123b>

The following code now determines the path to the Maven plugin.

```
123b  <AskConfig12 123b>≡ (123a)
      if [ ${MavenPluginFlag} -eq 1 ] &&[ -z ${MavenPluginPath} ]
      then
        DetectPlugins 'Maven' build-gbp-maven-plugin.sh
        if [ $? -eq 0 ]
        then
          MavenPluginPath=${PluginPath}
        else
          # If 'DetectPlugins' returns 24
          MavenPluginFlag=0
        fi
      fi
      changeflag=1
    fi
```

<AskConfig15 124>

April 6, 2025

When calling *DetectPlugins*, two options must be passed. Here *'Maven'* is passed as *\$1* to the variable *PluginName* and *build-gbp-maven-plugin.sh* is passed as *\$2* to the variable *PluginFile* (see chapter 29.1.2.1, page 120).

29.1.2.3. Variable query for Mozilla extensions

```
124  <AskConfig15 124>≡ (123b)
      # Webext Flag
      # This is needed to trigger special entries
      # in debian/control and debian/rules
      if [ -z "${WebextFlag}" ]
      then
          if whiptail --title "Mozilla AddOns" --defaultno \
              --yesno "Do you want to build a Mozilla AddOn package?" \
              --yes-button "Yes" --no-button "No" --defaultno 15 60
          <AskConfig15-1 125>
```

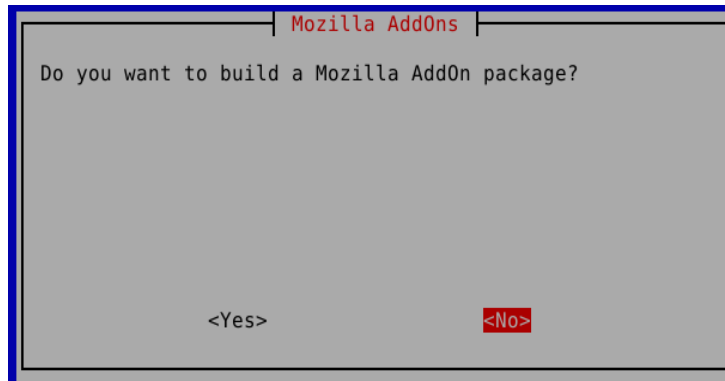



Figure 29.11.: Should an extension for Mozilla be packaged?

```

125  <AskConfig15-1 125>≡
      then
        WebextFlag=1
      else
        WebextFlag=0
      fi

      if [ ${WebextFlag} -eq 1 ] &&[ -z ${WebextPluginPath} ]
      then
        DetectPlugins 'Webext' build-gbp-webext-plugin.sh
        if [ $? -eq 0 ]
        then
          WebextPluginPath=${PluginPath}
        else
          # If 'DetectPlugins' returns 24
          WebextFlag=0
        fi
      fi
      changeflag=1
    fi
  <AskConfig18 126a>

```

(124)

29.1.2.4. Variable query for Python3 packages

A corresponding plugin can also be used for packaging **Python packages**. For this, the required information is first requested and entered into the configuration file..

```
126a  <AskConfig18 126a>≡ (125)
      # Python Flag
      # This is needed to trigger special entries
      # in debian/control and debian/rules
      if [ -z "${PythonFlag}" ]
      then
        if whiptail --title "Python3 programs" --defaultno \
          --yesno "Do you want to build a Python3 package?" \
            --yes-button "Yes" --no-button "No" --defaultno 15 60
        <AskConfig18-1 126b>
```

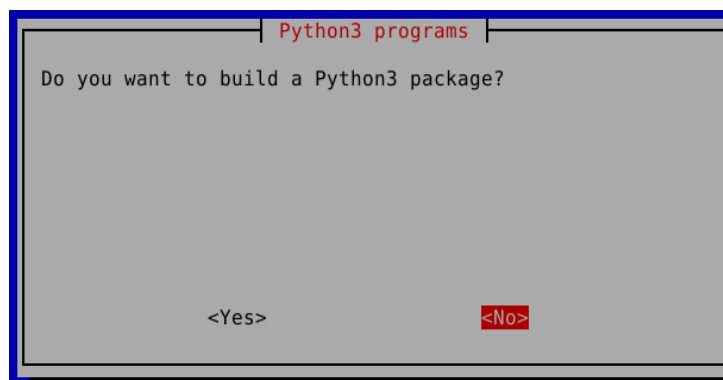


Figure 29.12.: Should a Python3 pact be built?

```
126b  <AskConfig18-1 126b>≡ (126a)
      then
        PythonFlag=1
      else
        PythonFlag=0
      fi
      <AskConfig19 127a>
```

In the following it is queried whether the path to the plugin is already known. If not, the *DetectPlugins* function is used to search for it. (Chapter 29.1.2.1, page 120)

```

127a  <AskConfig19 127a>≡ (126b)
      if [ ${PythonFlag} -eq 1 ] &&[ -z ${PythonPluginPath} ]
      then
        DetectPlugins 'Python3' build-gbp-python-plugin.sh
        if [ $? -eq 0 ]
        then
          PythonPluginPath=${PluginPath}
        else
          # If 'DetectPlugins' returns 24
          PluginFlag=0
        fi
      fi
      changeflag=1
    fi

    <AskConfig20 127b>

```

29.1.3. Saving the configuration

The (re-)creation of the configuration file is done in the following part of the *AskConfig* function.

First it is checked whether a configuration file already exists for the project. If it exists, it will be deleted.

The configuration file is a shell script. It therefore starts with a corresponding *Shebang*.

```

127b  <AskConfig20 127b>≡ (127a)
      if [ $changepflag -eq 1 ]
      then
        if [ -f ${ConfigPath}${OrigName} ]
        then
          rm ${ConfigPath}${OrigName}
        fi
        touch ${ConfigPath}${OrigName}

        # Shebang of the config file
        SB='#!/usr/bin/bash'

        # The config file is a shell script
        echo ${SB} >> ${ConfigPath}${OrigName}
        echo '# ConfigFile for '${OrigName}' >> ${ConfigPath}${OrigName}
        echo '## General parameters' >> ${ConfigPath}${OrigName}
        echo 'SourceName='${SourceName}' >> ${ConfigPath}${OrigName}
        echo 'PackName='${PackName}' >> ${ConfigPath}${OrigName}
        echo 'ProjectPath='${ProjectPath}' >> ${ConfigPath}${OrigName}
        echo 'SalsaName='${SalsaName}' >> ${ConfigPath}${OrigName}
    <AskConfig22 128a>

```

The following entries in the configuration file are only created if corresponding flags are set. This applies to `java` packages. An additional entry is made if they are built with *maven*. This also applies to extensions for `Firefox` and `Thunderbird` and programs in the `Python` programming language.

```
128a  <AskConfig22 128a>≡ (127b)
      echo '## Parameters for Java packages'>> ${ConfigPath}${OrigName}
      echo 'JavaFlag=${JavaFlag} >> ${ConfigPath}${OrigName}
      if [ ${JavaFlag} -eq 1 ]
      then
          echo 'MavenPluginFlag=${MavenPluginFlag} >> ${ConfigPath}${OrigName}
          if [ ${MavenPluginFlag} -eq 1 ]
          then
              echo 'MavenPluginPath=${MavenPluginPath} >> ${ConfigPath}${OrigName}
          fi
      fi

      echo '## Parameters for Webext packages'>> ${ConfigPath}${OrigName}
      echo 'WebextFlag=${WebextFlag} >> ${ConfigPath}${OrigName}
      if [ ${WebextFlag} -eq 1 ]
      then
          echo 'WebextPluginPath=${WebextPluginPath} >> ${ConfigPath}${OrigName}
      fi

      echo '## Parameters for Python3 packages'>> ${ConfigPath}${OrigName}
      echo 'PythonFlag=${PythonFlag} >> ${ConfigPath}${OrigName}
      if [ ${PythonFlag} -eq 1 ]
      then
          echo 'PythonPluginPath=${PythonPluginPath} >> ${ConfigPath}${OrigName}
      fi
      changeflag=0
  fi
}
```

<ReplaceTilde 129>

Since the tilde (~) in the path is not automatically replaced by `/home/<username>` when the script steps, this must be done by a *ReplaceTilde* function in the program script. Furthermore, any slash (/) at the end of the path is removed.

```
128b  <ConfigFileLEC6 128b>≡ (110)
      # Replace tilde if necessary
      SuspectPath=${ProjectPath}
      ReplaceTilde
      ProjectPath=${CleanPath}
```

<ConfigFileLEC7 130>

129 $\langle \text{ReplaceTilde 129} \rangle \equiv$ (128a)

```
function ReplaceTilde {
    # Called by ConfigFileLEC GbpConfIntegration
    RecentUser=$(whoami)
    tp=$(echo ${SuspectPath} | grep --count '~')
    if [ $tp -ge 1 ]
    then
        CleanPath=$(echo ${SuspectPath} | \
            sed --expression="s/~~/\home/${RecentUser}/g")
    else
        CleanPath=${SuspectPath}
    fi

    # Replace / at the end
    CleanPath=$(echo ${CleanPath} | sed --expression="s/\$/"/)
}
```

$\langle \text{ConfigFileLEC 109a} \rangle$

29.1.4. Example of a configuration file

```
#!/usr/bin/bash
# ConfigFile for <OrigName>
## General parameters
SourceName=<SourceName>
PackName=<PackName>
ProjectPath=/home/mechtilde/Projekte/Git/01_Salsa
SalsaName=<Name of the team>/<SourceName>.git
## Parameters for Java packages
JavaFlag=0
## Parameters for Webext packages
WebextFlag=0
## Parameters for Python3 packages
PythonFlag=0
RecentBranch=debian/sid
## Maintainer and Uploaders
Maintainer=<Name and E-Mail-Address of the Maintainer>
Uploaders=<Name and E-Mail-Address of the Uploaders>
## Download from upstream
DownloadURL=<Upstream URL for download>
RecentUpstreamSuffix=.xpi
# debian/sid_Dist=sid
```

29.2. Creating the infrastructure

The infrastructure of each project includes directories, a log file and a Git repository. If these are not already present, the necessary directories and the log file are created.

The infrastructure also includes a *chroot* directory *base.cow* or a *build chroot* directory. These are created only if they do not already exist for the distribution for which the package is to be built. (Chapter 34.5, page 295)

29.2.1. Definition of paths

At the end of the *ConfigFileLEC* function two more composite paths are defined.

```
130  <ConfigFileLEC7 130>≡ (128b)
      PrjPath=${ProjectPath}/${OrigName}
      GitPath=${PrjPath}/${SourceName}

    }

    <CreateDirsAndLogfile 131a>
```

29.2.2. Creating the necessary directories

First, the previously defined paths (chapter 29.2.1, page 130) are created.

```
131a  <CreateDirsAndLogfile 131a>≡ (130)
      function CreateDirsAndLogFile {
          # Called by BuildApp

          # Create directories if necessary
          mkdir --parents ${PrjPath} # redundantly?
          mkdir --parents ${GitPath}

      <CreateLogFile 131b>
```

29.2.3. Create log file

The date and time are initially entered in the log file each time the program is started.

```
131b  <CreateLogFile 131b>≡ (131a)
      # Create Log-File
      cd ${PrjPath}
      log=${PrjPath}/${OrigName}.log.txt
      touch ${log}
      echo -e "\n\n===\n=== $(date) ===\n\n">>${log}
  } }

  <InsertDebName 146a>
```

29.3. Git Repositories

For working with *git-buildpackage* a local Git repository with working directory is essential.

The program script assumes that both a local repository (chapter 19.4, page 76) and a repository on *salsa.debian.org* (chapter ??, page ??) should be created. In addition, the program script also takes into account a Git repository on its own Git server (chapter 19.4.2, page 76)..

Rebuilding a Git repository is the first step in building a new Debian package. After that, the source code for the (new) version is downloaded (chapter 31.3, page 182), a revision is built (chapter 32, page 225) and finally uploaded.(chapter 40, page 337). Therefore, the user is asked if he wants to build a new package.

29.3.1. Does a Git repository already exist?

As a precaution, it is checked whether a local Git repository already exists for the project to be created. It is considered if a Git repository exists in a parent directory.

If a local Git repository exists, the process continues with the selection of a Git branch (see chapter 30.4, page 163).

132a $\langle \text{BuildApp7 } 132a \rangle \equiv$ (105)

```
## Checks whether there is a git repo
cd ${GitPath}
git status 1>/dev/null 2>&1&
```

```
# '==' does the same as '-eq'
if [ $? == 0 ]
then
    if [ -d .git ]
    then
        SelectBranch
    else
```

$\langle \text{BuildApp8 } 132b \rangle$

If a Git repository exists in a parent directory, a notice will be provided.

132b $\langle \text{BuildApp8 } 132b \rangle \equiv$ (132a)

```
SupOrdMsg="Is there a git repository in a superordinate directory?"
echo ${SupOrdMsg} >> ${log}
whiptail --title="Attention!" --msgbox "${SupOrdMsg}" 15 60
StartTasks
```

```
fi
else
    StartTasks
fi
```

$\langle \text{BuildApp10 } 157 \rangle$

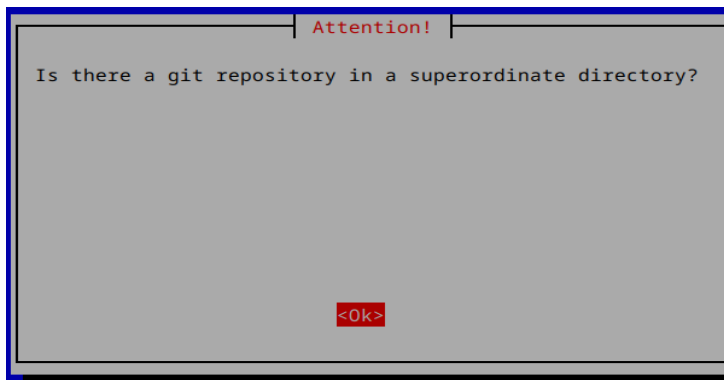


Figure 29.13.: Is there a parent Git repository?

If no `Git` repository exists in the corresponding directory, the *StartTasks* function is called.

29.3.2. Selection dialog

If no `Git` repository exists yet, the script provides three possibilities to create such a repository. These are the new creation with *git init* and the cloning of an existing (`Git`) repository from *salsa.debian.org* (chapter 29.5, page 149). The latter requires above all that a branch *prostine-tar* exists there.

Otherwise, the repository can be created by *gbp import-dsc* (chapter 29.6, page 155).

```
133 <StartTasks 133>≡
    function StartTasks {
        # Called by BuildApp

        Task=$(whiptail --title "Tasks for building a new package:" \
            --radiolist "What do you like to do to build a new package? " 17 60 9 \
                "0" "Create a git repo and download upstream code" on \
                "11" "Clone an existing repo from Salsa" off \
                "12" "Importing already existing Debian packages" off \
            --cancel-button "Exit" 3>&2 2>&1 1>&3)

        <StartTasks1 134>
```

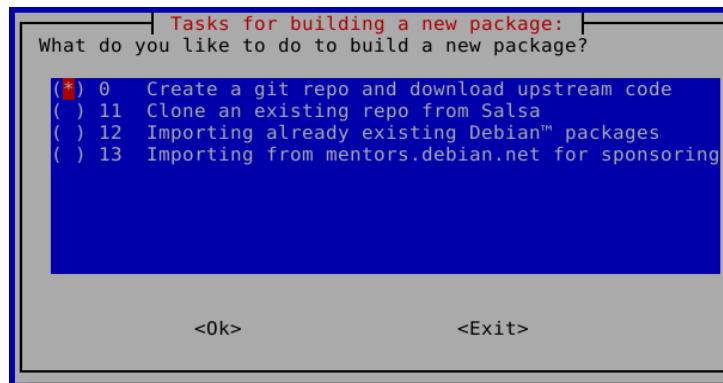


Figure 29.14.: Create a new package

134 $\langle \text{StartTasks1 } 134 \rangle \equiv$ (133)

```

if [ -z "${Task}" ]
then
    exit
fi
TaskSelect
}

```

$\langle \text{DebDiff } 319 \rangle$

The items of the menu deal with different ways to set up a new project. It is mainly about obtaining the source code of the upstream project.

The call of the corresponding functions is done by the function *TaskSelect*.

```

135  <TaskSelect 135>≡
    function TaskSelect {
        # Called by StartTasks CommonTasks

        ## The lines below serve also the sequence control. A called function
        ## changes finally the variable 'Task', so one of the following
        ## if-clauses matches. That is why the following lines can not be
        ## replaced by a case statement.

        ## Start tasks

        # Building a new package from archive
        if [ $Task -eq 0 ]
        then
            BuildNewPackage
        fi

        # Clone an existing repo from Salsa
        if [ $Task -eq 11 ]
        then
            CloneFromSalsa
        fi

        # Importing already existing Debian package
        if [ $Task -eq 12 ]
        then
            ImportDebianPackage
        fi

    <TaskSelect3 172>

```

When *Build a new package from archive*, the local `Git` repository is created first (chapter 29.4, page 136). Then a new version is downloaded (chapter 31.4, page 185).

When *Clone an existing repo from Salsa*, an existing repository is downloaded from *salsa.debian.org* (chapter 29.5, page 149).

At *Importing already existing DebianTM package* a **.dsc* file is downloaded and a `Git` repository is created by the *gbp import-dsc* program (chapter 29.6, page 155)..

29.4. Creating a new local `Git` repository

One way to build a `Git` repository is to create a new one using *git init*. This is done in the *BuildNewPackage* function. Before this a corresponding entry is made in the log file.

```
136a  <BuildNewPackage 136a>≡ (148b)
      function BuildNewPackage {
          # Called by TaskSelect
          cd ${GitPath}
          if [ -d .git ]
          then
              echo "There seems already to be a git repository in ${GitPath}." >> ${log}
              if ! whiptail --title "Warning" \
                  --yesno "There seems already to be a git repository in ${GitPath}.\n \
                  However continue?" --yes-button "Yes" --no-button "No" 15 60
              then
                  echo "Exit" >> ${log}
                  exit
              fi
          else
              echo "In ${GitPath} a new git repository will be created." >> ${log}
              git init
          <BuildNewPackage2 147a>
      }
```

29.4.1. Add name and email address to `Git` repository

```
136b  <BuildNewPackage3 136b>≡ (147a)
      if whiptail --title "Name and email" \
          --yesno "Do you like to add your name and email address \n \
          to the local git config file?" --yes-button "Yes" \
          --no-button "No" 15 60
      then
          AddNameAndEmail
      fi
      <BuildNewPackage5 148a>
```

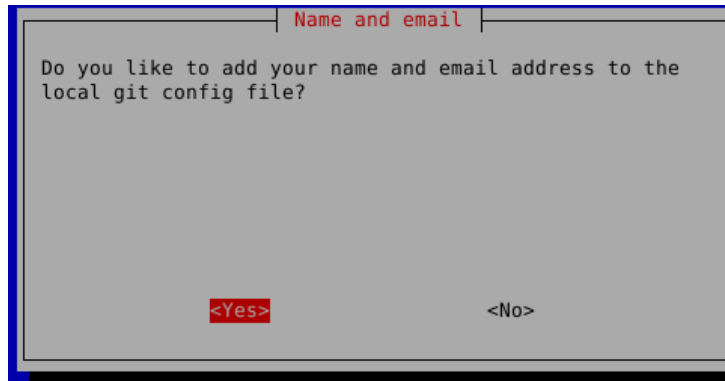


Figure 29.15.: Name and email.

137a $\langle \text{AddNameAndEmail 137a} \rangle \equiv$ (146b)

```
function AddNameAndEmail {
    # Called by BuildNewPackage CloneFromSalsa
    # ImportDebianPackage and itself

    DEBValues
    GCName=${DEBFULLNAME}
    GCEmail=${DEBEMAIL}
```

$\langle \text{AddNameAndEmail1 145} \rangle$

So that the packager does not always have to write his full name and e-mail address, the script looks for this data first in the configuration file and then in the `~/.bashrc`. If necessary, what is found or asked for is entered into the configuration file.

137b $\langle \text{Bashrc 137b} \rangle \equiv$

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

```

138  <DEBValues 138>≡ (139)
    function DEBValues {
        # Called by DebianControlTemplate PatchHeader AddNameAndEmail
        MaintainerF=0

        # if Maintainer is stored in config file
        if [ ${Maintainer} ]
        then
            Maintainer=$(echo ${Maintainer} | sed --expression='s/_/ /g')
            Maintainer=$(echo ${Maintainer} | sed --expression='s/@lt@/</g')
            Maintainer=$(echo ${Maintainer} | sed --expression='s/@gt@/>/g')
            DEBFULLNAME=$(echo ${Maintainer} | sed --expression='s/<.*//')
            DEBEMAIL=$(echo ${Maintainer} | sed --expression='s/^.*</' | sed 's/>/'')
            MaintainerF=1
        fi

        # if Uploaders is stored in config file
        if [ ${Uploaders} ]
        then
            Uploaders=$(echo ${Uploaders} | sed --expression='s/_/ /g')
            Uploaders=$(echo ${Uploaders} | sed --expression='s/@lt@/</g')
            Uploaders=$(echo ${Uploaders} | sed --expression='s/@gt@/>/g')
            DEBFULLNAME=$(echo ${Uploaders} | sed --expression='s/<.*//')
            DEBEMAIL=$(echo ${Uploaders} | sed --expression='s/^.*</' | sed 's/>/'')
        fi

        # Looking for a team as maintainer
        if [ ${MaintainerF} -eq 0 ]
        then
            TeamMaintainer
        fi
    }
<DEBValues3 140>

```

There are packages that are managed by a team. Usually, many similar packages are maintained by such a team. In these cases the Debian project member acts as *uploader* and the team as *maintainer*. This is entered accordingly in the *debian/control* file.

```

139 <TeamMaintainer 139>≡ (234a)
function TeamMaintainer {
    # Called by DEBValues

    # Makes sure that variable exists
    if [ -z '${JavaFlag}' ]
    then
        JavaFlag = 0
    fi

    if [ ${JavaFlag} -eq 1 ]
    then
        Maintainer="Debian Java Maintainers \
        <pkg-java-maintainers@lists.alioth.debian.org>"
        MaintainerF=1
    fi

    if [ -z '${WebextFlag}' ]
    then
        WebextFlag = 0
    fi

    if [ ${WebextFlag} -eq 1 ]
    then
        Maintainer="Debian Mozilla Extension Maintainers \
        <pkg-mozext-maintainers@alioth-lists.debian.org>"
        MaintainerF=1
    fi

    if [ -z '${PythonFlag}' ]
    then
        PythonFlag = 0
    fi

    if [ ${PythonFlag} -eq 1 ]
    then
        Maintainer="Debian Python Team <team+python@tracker.debian.org>"
        MaintainerF=1
    fi
}

<DEBValues 138>

```

```

140  <DEBValues3 140>≡ (138)
    # Extracts DEBFULLNAME and DEBEMAIL from ~/.bashrc (if exist)
    if [ ${MaintainerF} -eq 0 ]
    then
        if grep --quiet 'DEBFULLNAME' ~/.bashrc
        then
            dfnb=$(grep DEBFULLNAME ~/.bashrc)
            dfnb=$(echo ${dfnb} | sed --expression='s/export .*//')
            dfnb=$(echo ${dfnb} | sed --expression='s/DEBFULLNAME=//')
            dfnb=$(echo ${dfnb} | sed --expression='s/"//g')
            dfnb=$(echo ${dfnb} | sed --expression="s/'//g")
            DEBFULLNAME=${dfnb}
        fi
        if grep --quiet 'DEBEMAIL' ~/.bashrc
        then
            demb=$(grep 'DEBEMAIL' ~/.bashrc)
            demb=$(echo ${demb} | sed --expression='s/export .*//')
            demb=$(echo ${demb} | sed --expression='s/DEBEMAIL=//')
            demb=$(echo ${demb} | sed --expression='s/"//g')
            demb=$(echo ${demb} | sed --expression="s/'//g")
            DEBEMAIL=${demb}
        fi
        Maintainer=${DEBFULLNAME}" <${DEBEMAIL}">"
        MaintainerF=1
    fi

    # Insert name and email address
    if [ ${MaintainerF} -eq 0 ]
    then
        DEBFULLNAME=$(whiptail --title "Name of the maintainer" \
            --inputbox "Please insert full name of the maintainer" \
            --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
        DEBEMAIL=$(whiptail --title "Email of the maintainer" \
            --inputbox "Please insert email address of the maintainer" \
            --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
        Maintainer=${DEBFULLNAME}" <${DEBEMAIL}">"
        changeflag=1
        MaintainerF=1
    fi

    if ! whiptail --title "Maintainer" \
        --yesno "The full name and email address of the maintainer(s):\n \
            ${Maintainer}" --yes-button "Yes" --no-button "No" 15 60
    <DEBValues5 141a>

```

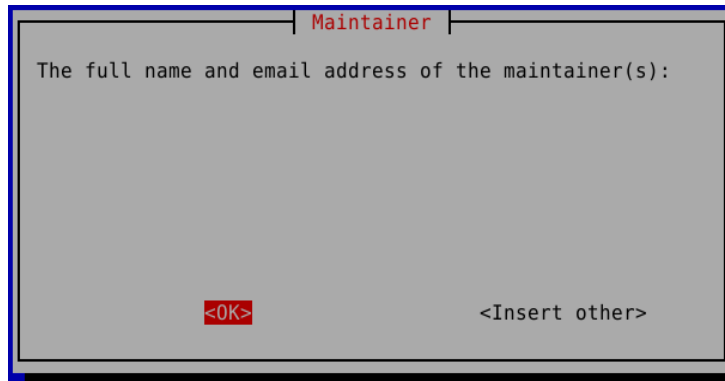



Figure 29.16.: Debian-Maintainer OK?

141a $\langle \text{DEBValues5 } 141a \rangle \equiv$ (140)
 then
 DEBFULLNAME=\$(whiptail --title "Name of the maintainer" \
 --inputbox "Please insert full name of the maintainer" \
 --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
 $\langle \text{DEBValues5-1 } 141b \rangle$

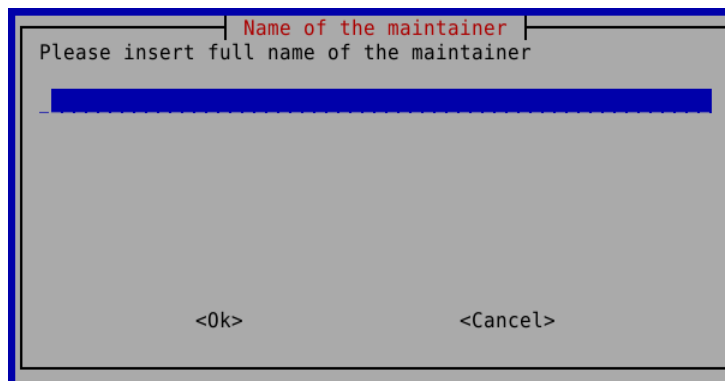


Figure 29.17.: Name of the Debian-Maintainer

141b $\langle \text{DEBValues5-1 } 141b \rangle \equiv$ (141a)
 DEBEMAIL=\$(whiptail --title "Email of the maintainer"\
 --inputbox "Please insert email address of the maintainer" \
 --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
 $\langle \text{DEBValues5-2 } 142a \rangle$

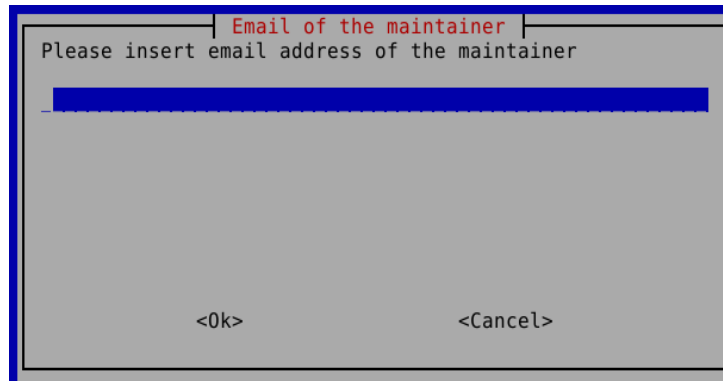


Figure 29.18.: Email of the Debian-Maintainer

```

142a  <DEBValues5-2 142a>≡                                     (141b)
      Maintainer=${DEBFULLNAME}" <"${DEBEMAIL}">"
      changeflag=1
      MaintainerF=1
      fi
      <DEBValues6 142b>

142b  <DEBValues6 142b>≡                                     (142a)

      # Insert maintainer data into config file if necessary
      if [ $changeflag -eq 1 ]
      then
        # Because Maintainer contains blanks
        MaintainerCF=$(echo ${Maintainer} | sed --expression='s/ /_/g')
        # Remove < and >
        MaintainerCF=$(echo ${MaintainerCF} | sed --expression='s/</@lt@/g')
        MaintainerCF=$(echo ${MaintainerCF} | sed --expression='s/>/@gt@/g')
        echo '## Maintainer and Uploaders' >> ${ConfigPath}${OrigName}
        echo 'Maintainer='${MaintainerCF} >> ${ConfigPath}${OrigName}
        changeflag=0
      fi

      <DEBValues7 143a>

```

143a $\langle DEBValues7\ 143a \rangle \equiv$ (142b)

```

# Insert uploaders data into config file
if necessary if [ ${JavaFlag} -eq 1 ]
then
  if [ -z "${Uploaders}" ]
  then
    if grep --quiet 'DEBFULLNAME' ~/.bashrc
    then
      dfnb=$(grep DEBFULLNAME ~/.bashrc)
      dfnb=$(echo ${dfnb} | sed --expression='s/export .*/')
      dfnb=$(echo ${dfnb} | sed --expression='s/DEBFULLNAME=//')
      dfnb=$(echo ${dfnb} | sed --expression='s/"//g')
      dfnb=$(echo ${dfnb} | sed --expression="s/'//g")
      DEBFULLNAME=${dfnb}
    else
      DEBEMAIL=$(whiptail --title "Email of the uploader" \
        --inputbox "Please insert email address of the uploader" \
        --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&
    fi
  fi
 $\langle DEBValues8\ 143b \rangle$ 

```

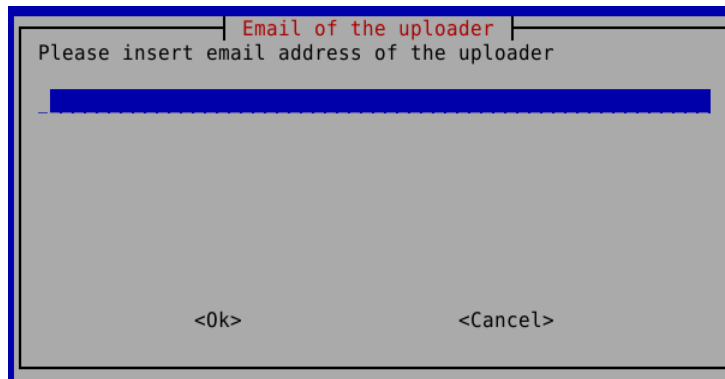


Figure 29.19.: Email of the Debian-Uploaders

143b $\langle DEBValues8\ 143b \rangle \equiv$ (143a)

```

if grep --quiet 'DEBEMAIL' ~/.bashrc
then
  demb=$(grep 'DEBEMAIL' ~/.bashrc)
  demb=$(echo ${demb} | sed --expression='s/export .*/')
  demb=$(echo ${demb} | sed --expression='s/DEBEMAIL=//')
  demb=$(echo ${demb} | sed --expression='s/"//g')
  demb=$(echo ${demb} | sed --expression="s/'//g")
  DEBEMAIL=${demb}
else
  DEBEMAIL=$(whiptail --title "Email of the uploader" \
    --inputbox "Please insert email address of the uploader" \
    --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
fi
 $\langle DEBValues9\ 144 \rangle$ 

```

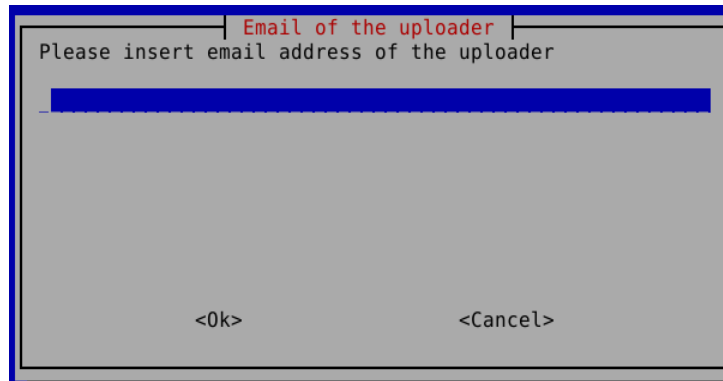


Figure 29.20.: Email of the Debian-Uploaders

```

144  <DEBValues9 144>≡ (143b)
      Uploaders=${DEBFULLNAME}" <"${DEBEMAIL}">"
      # Because Uploaders contains blanks
      UploadersCF=$(echo ${Uploaders} | sed --expression='s/ /_/g')
      # Remove < and >
      UploadersCF=$(echo ${UploadersCF} | sed --expression='s/</@lt@/g')
      UploadersCF=$(echo ${UploadersCF} | sed --expression='s/>/@gt@/g')
      echo 'Uploaders='${UploadersCF} >> ${ConfigPath}${OrigName}
      changeflag=0
    fi
  fi
}

<DebianControlTemplate 234b>

```

145 $\langle \text{AddNameAndEmail1 145} \rangle \equiv$ (137a)

```

    if [ -z "${GCName}" ]
    then
        InsertDebName
    fi
    # if [ -n "${GCName}" ]
    # then
        git config user.name ${GCName}
    # fi
    if [ -z "${GCEmail}" ]
    then
        InsertDebEmail
    fi
    # if [ -n "${GCEmail}" ]
    # then
        git config user.email ${GCEmail}
    # fi

    configStr=$(git config --list | grep 'user')

    if ! whiptail --title "Result" \
        --yesno "${configStr}\nAllright?" \
        --yes-button "Yes" --no-button "No" 15 60
    then
        AddNameAndEmail
    fi
}

 $\langle \text{AddHomeServer 148b} \rangle$ 

```

April 6, 2025

```
146a  <InsertDebName 146a>≡ (131b)
      function InsertDebName {

          # Called by AddNameAndEmail and itself
          DEBFULLNAME=$(whiptail --title "Name of the maintainer" \
            --inputbox "Please insert full name of the maintainer" \
            --cancel-button "Exit" 15 60 3>&2 2>&1 1>&3)

          if [ $? -ne 0 ]
          then
              exit
          fi

          # Test Name
          if [ -z "${DEBFULLNAME}" ]
          then
              whiptail --title "Your Name" \
                --msgbox "Your name is neccessary." 15 60
              InsertDebName
          fi
      }
      <InsertDebEmail 146b>

146b  <InsertDebEmail 146b>≡ (146a)
      function InsertDebEmail {

          # Called by AddNameAndEmail and itself
          DEBEMAIL=$(whiptail --title "Email of the maintainer" \
            --inputbox "Please insert email address of the maintainer" \
            --cancel-button "Exit" 15 60 3>&2 2>&1 1>&3)

          if [ $? -ne 0 ]
          then
              exit
          fi

          # Regex string
          EmailR="\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,6}\b"
          # Test email address
          if ! echo ${DEBEMAIL} | grep -E ${EmailR} > /dev/null
          then
              whiptail --title "Bad!" --msgbox "That's no email address." 15 60
              InsertDebEmail
          fi
      }
      <AddNameAndEmail 137a>
```



Figure 29.21.: Debian maintainer data

29.4.2. Repository at *salsa.debian.org*

The program script adds the Salsa repository as a “Remote repository”.

```
147a <BuildNewPackage2 147a>≡ (136a)
      git remote add salsa git@salsa.debian.org:${SalsaName}

<BuildNewPackage3 136b>
```

29.4.2.1. Manually

The corresponding repository on *salsa.debian.org* is then created manually (chapter ??, page ??) (see chapter 47.1, page 387).

The program script also reminds the user of this.

```
147b <BuildNewPackage6 147b>≡ (148a)

      whiptail --title "If not happened yet:" \
      --msgbox "Please create a git repo ${SalsaName} \n \
      on salsa.debian.org!" 15 60
      Task=2 # Go to BuildNewVersion
    }

<IdentifyBranches 152a>
```

29.4.2.2. Within the Java team

Within the Java team, a new repository should be created using the *setup script* provided by the team (chapter ??, page ??). To facilitate this, the build script will automatically trigger this.

- script copies setup script from *.debian_project*
- script tells how to generate token
- script queries token (command line)
- script writes in setup the token
- script calls setup script, with source name

29.4.3. Display remote server

The remote servers entered in the Git repository are displayed. A custom server is displayed only if it has been entered previously.

```
148a  <BuildNewPackage5 148a>≡ (136b)
      AddHomeServer
      fi
      <BuildNewPackage6 147b>

148b  <AddHomeServer 148b>≡ (145)
      function AddHomeServer {
        # Called by BuildNewPackage AddGitServer ImportDebianPackage
        if [ -n "$ServerName" ]
        then
          git remote add home $(whoami)@${ServerName}:/srv/git/${SourceName}.git

          whiptail --title "Remoteserver" \
            --msgbox "New server added:\n$(git remote --verbose)" 15 60
          echo -e "New server added:\n $(git remote --verbose)" >> ${log}
        fi
      }

      <BuildNewPackage 136a>
```




Figure 29.22.: Add remote server

Then a new version is downloaded (chapter 31.4, page 185).

29.5. Clonen from *salsa.debian.org*

In this case, some of the steps mentioned in the other chapters are omitted. The cloning is done with the command *gbp clone*. After that, the remote repository is renamed from *origin* to *salsa* on successful cloning. This allows for more meaningful naming of the repositories.

Otherwise, an error message is displayed and the program ends.

```

149a  <CloneFromSalsa 149a>≡ (176)
      function CloneFromSalsa {
          # Called by TaskSelect
          echo "Clone an existing repo from salsa.debian.org" >> ${log}
          cd ${PrjPath}
          gbp clone git@salsa.debian.org:${SalsaName} --aliases ${SourceName}
          if [ $? -eq 0 ]
          then
              cd ${GitPath}
              git remote rename origin salsa
              echo "${SalsaName} was cloned" >> ${log}
          else
              echo "${SalsaName} could not be cloned"
              exit
          fi
      }

      <CloneFromSalsa2 149b>

```

If the download is successful, the *DebianBranchName* function displays the downloaded branches and the active branch is displayed as a sound.

```

149b  <CloneFromSalsa2 149b>≡ (149a)
      # Identify branches and choose one
      DebianBranchName

      <CloneFromSalsa3 153b>

```

29.5.1. Determination of the Git branches

The *DebianBranchName* function is used to determine the Git branches of the cloned Salsa repository.

To do this, first execute the *IdentifyBranches* function (chapter 29.5.2, page 152).

```
150a  <DebianBranchName 150a>≡ (153a)
      function DebianBranchName {
          # Called by CloneFromSalsa

          ## Identify and show branches
          IdentifyBranches
          ba=($bl)

          whiptail --title "Branches in repo ${OrigName}:" --msgbox "${bl}" 15 60
```

<DebianBranchName2 150b>

Then the active Git branch is determined.

```
150b  <DebianBranchName2 150b>≡ (150a)
      for element in ${ba[*]}
      do
          # Find the default branch
          if echo ${element} | grep --quiet '^x_'
          then
              DefaultBranch=$(echo ${element} | sed --expression='s/^x_//')
              whiptail --title "Recent branch found" --msgbox "Found: ${DefaultBranch}" 15 60
          fi
      done
      <DebianBranchName3 151>
```

```

151  <DebianBranchName3 151>≡ (150b)
      # Ignore HEAD
      if echo ${element} | grep --quiet 'HEAD'
      then
          continue
      fi
      # Checkout all branches
      if echo ${element} | grep --quiet '^remotes/salsa/'
      then
          NewBranchName=$(echo ${element} | \
          sed --expression='s/^remotes\/salsa\/\\\\')
          git checkout ${NewBranchName}
          whiptail --title "Checkout branch" \
          --msgbox "Checkout of ${NewBranchName}" 15 60
      fi
  done
  # Finally checkout the default branch (again)
  git checkout ${DefaultBranch}
  whiptail --title "Checkout branch" \
  --msgbox "Checkout of ${DefaultBranch}" 15 60

  # Insert default branch into the config file and write into logfile
  echo 'DefaultBranch='${DefaultBranch} >> ${ConfigPath}${OrigName}
  echo "Branches: "${bl} >> ${log}
  echo "DefaultBranch: "${DefaultBranch} >> ${log}
  RecentBranch=${DefaultBranch}
  whiptail --title "Please check! (1)" \
  --msgbox "The branch is ${RecentBranch}" 15 60
  echo 'RecentBranch='${RecentBranch} >> ${ConfigPath}${OrigName}
  echo "RecentBranch: "${RecentBranch} >> ${log}
  Distro4Branch
}

```

<FailureNotice 165>

April 6, 2025

Finally, by calling the *Distro4Branch* function, a distribution is assigned to the **Git** branch (chapter 29.5.3, page 152)

29.5.2. Git branches detect

This function determines all existing branches. This is done with the command *git branch --all*. When marking the active branch, asterisks (*) and spaces are replaced by small X (x) and Unterstrich(_).

This function is called at various points in the step of the program script.

```
152a  <IdentifyBranches 152a>≡ (147b)
      function IdentifyBranches {
          # Called by DebianBranchName AskDist DebianBranches
          cd ${GitPath}
          # sed is used to kill the asterisk
          bl=$(git branch --all | sed --expression='s/* /x_/')
      }

      <CreateSchroot 295b>
```

29.5.3. Assign Git Branch Distribution

At various points in the program flow it is necessary to assign a Debian release to a **Git** branch.

```
152b  <Distro4Branch 152b>≡ (298b)
      function Distro4Branch {
          # Called by DebianBranchName CreateNewBranch AskDist BuildNewRevision

          # Set Debian distribution for branch
          if [ ${RecentBranch} != "debian/sid" -o -z "${RecentBranchD}" ]
          then
              CowL=$(ls /var/cache/pbuilder/ | grep .cow | grep '-' | \
                  sed 's/base-//' | sed 's/.cow//') i
              CowA=( $CowL )

              i=1; cowE="0 sid on "
              for element in ${CowA[*]}
              do
                  cowE=$cowE' '$i' '${element}' off '
                  i=$((expr $i + 1))
              done

              RecentBranchDNr=$(whiptail --title "Debian release" \
                  --radiolist "Select pbuilder cow:" \
                  --cancel-button "Other" 15 60 8 \ $cowE 3>&2 2>&1 1>&3)
          <Distro4Branch2 153a>
```

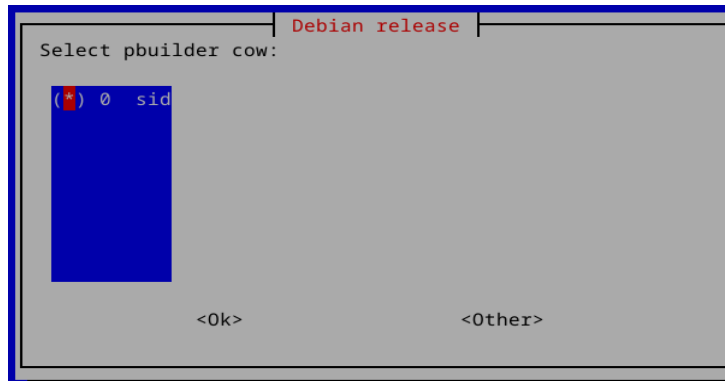


Figure 29.23.: Choosing the Debian Release.

```

153a  <Distro4Branch2 153a>≡ (152b)
      if [ $? -eq 1 ]
      then
        CreateNewCow
      fi

      if [ ${RecentBranchDNr} -eq 0 ]
      then
        bDist="sid" else RecentBranchDNr=$(expr ${RecentBranchDNr} - 1)
        bDist=${CowA[${RecentBranchDNr}]}
      fi

      echo "# "${RecentBranch}"_Dist="${bDist} >> ${ConfigPath}${OrigName}
      RecentBranchD=${bDist}
      echo "Notice from Distro4Branch: The distribution is " \
        ${RecentBranchD} >> ${log}
    fi
  }

  <DebianBranchName 150a>

```

29.5.4. Add name and email address

```

153b  <CloneFromSalsa3 153b>≡ (149b)
      if whiptail --title "Name and email" \
        --yesno "Do you like to add your name and email address \n \
        to the local git config file?" --yes-button "Yes" \
        --no-button "No" 15 60
      <CloneFromSalsa5 154a>

```

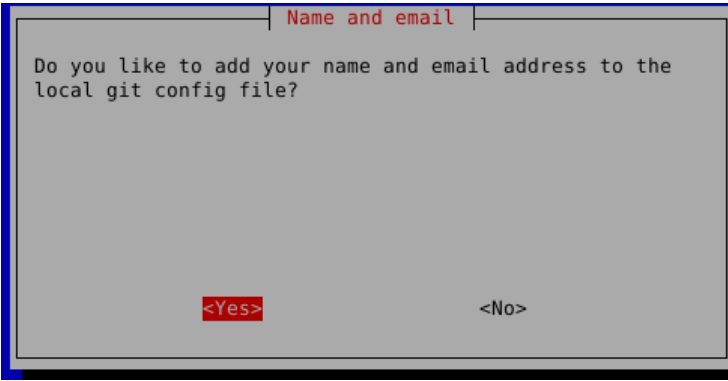


Figure 29.24.: Name and email.

154a $\langle \textit{CloneFromSalsa5} \ 154a \rangle \equiv$ (153b)
 then
 AddNameAndEmail
 fi
 $\langle \textit{CloneFromSalsa7} \ 154b \rangle$

154b $\langle \textit{CloneFromSalsa7} \ 154b \rangle \equiv$ (154a)
 AddHomeServer
 PQImport 1
 CommonTasks
 }
 $\langle \textit{CutSuffix} \ 191a \rangle$

29.6. Import of a Debian package

Not all packages of the Debian project are located on *salsa.debian.org*, a Gitlab instance. Of these, not all packages are built with *git-buildpackage* either. Sometimes the *pristine-tar* branch is missing. In this case, cloning does not make sense.

These packages can also be built using this program script. In the file */etc/apt/sources.list* or in a file in the directory */etc/apt/sources.list.d* the entry.

```
deb-src http://deb.debian.org/debian/ sid main
```

activated must be present. This means that after activation, *sudo apt update* must also be executed.

For rare special cases, adapt this line accordingly. This enables the download of source packages.

```
155a <ImportDebianPackage 155a>≡ (321)
function ImportDebianPackage {
    # Called by TaskSelect

    echo "Import an existing package without pristine-tar" >> ${log}
    cd ${PrjPath}
    echo "Download ${SourceName} with apt source" >> ${log}
    apt source --download-only ${SourceName} >> ${log}
```

```
<ImportDebianPackage1 155b>
```

With *apt source --download-only* only the source package is downloaded.

Creating the Git directory is done with *gbp import-dsc* and the downloaded **.dsc* file.

```
155b <ImportDebianPackage1 155b>≡ (155a)
    cd ${GitPath}
    git init
    git remote add salsa git@salsa.debian.org:${SalsaName}
    GpgKeyAvailable
```

```
<ImportDebianPackage2 155c>
```

Since the *gbp import-dsc* function performs a signing, the *GpgKeyAvailable* function (chapter 29.7, page 156) first queries whether the GnuPG key is available.

```
155c <ImportDebianPackage2 155c>≡ (155b)
    echo "gbp import-dsc" >> ${log}
    gbp import-dsc --verbose ../${SourceName}*.dsc . >> ${log}
```

```
<ImportDebianPackage3 156a>
```

April 6, 2025

This is followed by displaying the existing `Git` branches and determining the active branch by calling the *DebianBranchName* function (chapter 29.5.1, page 150).

```
156a  <ImportDebianPackage3 156a>≡ (155c)
      # Identify branches and choose one DebianBranchName

      if whiptail --title "Name and email" \
        --yesno "Do you like to add your name and email address \n \
        to the local git config file?" --yes-button "Yes" \
        --no-button "No" 15 60
      then
        AddNameAndEmail
      fi

      <ImportDebianPackage4 156b>

156b  <ImportDebianPackage4 156b>≡ (156a)
      AddHomeServer
      PQImport
      CommonTasks

    }

    <CommonTasks 171>
```

29.7. GnuPG Key available?

The *GpgKeyAvailable* function prompts the user to check if the **GnuPG** key is available for signing.

It is called at various points in the program script. If the question about the availability of the **GnuPG** key is answered in the negative, the program script terminates.

```
156c  <GpgKeyAvailable 156c>≡ (162b)
      function GpgKeyAvailable {
        # Called by BuildWithUscan Import2Git CreateSignature ImportDebianPackage
        # PrepareUploading

        if ! whiptail --title "GPG-Key available?" \
          --yesno "GPG-Key must be available for signing." \
          --yes-button "Yes, is available" --no-button "Exit" 15 60
        then
          exit
        fi
        GettingFingerprint
      }

      <DetectPlugins 120>
```

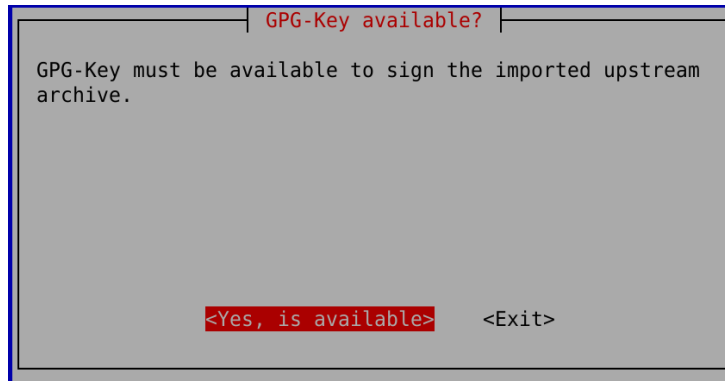



Figure 29.25.: GPG-Key available

29.8. Starting the packaging process

The *BuildApp* function calls the task selection for each step of packaging at the end.

```

157  ⟨BuildApp10 157⟩≡
      # Start of Packaging
      CommonTasks
    }
    ⟨MainProgram 104a⟩
  (132b)

```


30. Work in an created projectt

30.1. Load and edit configuration file

If a project name is entered, the program script tries to load the configuration file of this project and display it with *less*.

The function *ConfigFileLEC* shows the loaded configuration file and then asks if it is correct. Then it can be decided if this file should be edited.

```
159 <ConfigFileLEC1 159>≡ (109a)
    if [ -f ${ConfigPath}${OrigName} ]
    then
        . ${ConfigPath}${OrigName} # executes config script
        # whiptail --title "Config file found" \
        # --msgbox "${ConfigPath}${OrigName} was loaded." 15 60
        less --LINE-NUMBERS ${ConfigPath}${OrigName}
        if ! whiptail --title "Check config file" \
        --yesno "Is the config file OK?" \
        --yes-button "Yes" --no-button "No" 15 60
    <ConfigFileLEC2 160>
```

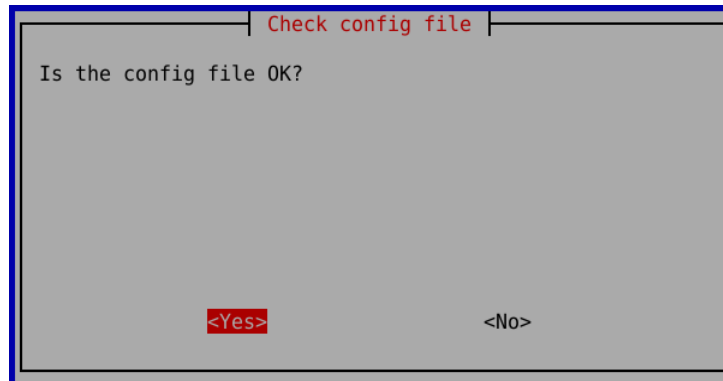


Figure 30.1.: Query - configuration file

If the question whether the configuration file is correct is answered in the negative, it can be edited. This can be done with the editor *nano* or by answering questions.

Otherwise, continue with chapter 30.4, page 163. However, if only one or no branch exists, continue with chapter 30.4.7, page 170.

```
160  <ConfigFileLEC2 160>≡ (159)
      then
        Edit=$(whiptail --title "Edit Config File" \
          --radiolist "How do you like to edit the config file?" \
          15 60 2 "0" "Using Nano" on \
          "1" "Answering questions" off \
          --cancel-button "Exit" 3>&2 2>&1 1>&3)
      <ConfigFileLEC3 161>
```

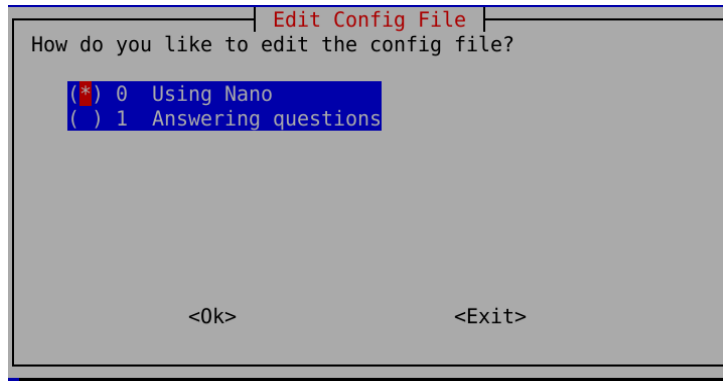


Figure 30.2.: Query - edit configuration file.

If no edit method is specified, the program exits.

161 $\langle \text{ConfigFileLEC3 } 161 \rangle \equiv$ (160)

```

if [ -z "${Edit}" ]
then
    whiptail --title "Bye" --msgbox "Bye" 15 60
    exit
fi
if [ ${Edit} -eq 0 ]
then
    nano --linenums --mouse \
        --softwrap ${ConfigPath}${OrigName}
    . ${ConfigPath}${OrigName}
else
    AskConfig
fi
fi
 $\langle \text{ConfigFileLEC4 } 109b \rangle$ 

```

If the configuration file is to be edited with the editor *nano*, this is called and opens. After the termination of the same (with *STRG-X*) the program script runs further (chapter 30.4, page 163).

If the adjustment is to be done by means of queries, the function *AskConfig* (see chapter 29.1.1, page 111) is called. After that the file is rewritten.

30.2. Modify lines in the configuration file

The script also contains a function that allows to change individual values in the configuration file. For this purpose, two parameters are passed to the function, namely firstly the identifier of the variable and secondly the new value of the same.

This function is used by the *BuildNewVersion* (chapter 31.4.2, page 185), *PQMmigration* (chapter 33.1, page 254), *ChangeEntry* (chapter 30.4.5, page 168) and *OwnServer* (chapter 41.2, page 356) functions.

```
162a  <ReplaceConfigLines 162a>≡ (163a)
      function ReplaceConfigLines {
          # Called by BuildNewVersion PQMigration ChangeEntry OwnServer CurSuffix

          # This function needs two parameters:
          # First the name of the variable
          # and second it's new value
          ConfProp=$1
          ConfVal=$2
```

```
<ReplaceConfigLines1 162b>
```

The function checks if the line to be changed exists in the configuration file. If not, it is created with the *InsertConfigLines* function.

Before replacing with *sed*, mask (also with *sed*) any slashes that may be present in the variable *ConfVal*.

```
162b  <ReplaceConfigLines1 162b>≡ (162a)
      cprop=$(grep --count ${ConfProp} ${ConfigPath}${OrigName})
      if [ ${cprop} -ge 1 ]
      then
          # Masquerade slashes
          ConfVal=$(echo ${ConfVal} | sed 's/\//\\//g')

          sed --in-place --expression="s/${ConfProp}=.*/${ConfProp}=${ConfVal}/g" \
            ${ConfigPath}${OrigName}
      else
          # InsertConfigLine needs two parameters:
          # name of the variable and new value
          InsertConfigLine ${ConfProp} ${ConfVal}
      fi
  }
```

```
<GpgKeyAvailable 156c>
```

30.3. Insert line into configuration file

The *InsertConfigLine* function inserts a line into the configuration file. It must be passed the same parameters as the *ReplaceConfigLines* function.

```
163a <InsertConfigLine 163a>≡
    function InsertConfigLine {
        # Called by ReplaceConfigLines

        # This function needs two parameters:
        # First the name of the variable
        # and second it's value

        ConfProp=$1
        ConfVal=$2

        echo "${ConfProp}"="${ConfVal}" >> ${ConfigPath}${OrigName}
    }

    <ReplaceConfigLines 162a>
```

30.4. Selecting a Git branch

To allow switching the *Git* branch, a check is made to see if there are any unversioned changes in the current branch. This check is done with the *CheckGitStatus* function (chapter 30.4.1, page 164).

If no change of the *Git* branch is required, changes in the *debian/* directory do not necessarily need to be versioned.

Changes already made to the upstream code must be restored using *git restore*.

If several branches exist, one branch can be selected. If only one branch exists, it will be named. Otherwise the function *StartTasks* is called (chapter 29.3, page 132).

This function can be called from the task selection (chapter 30.5, page 171).

The *SelectBranch* function is also called when building a new revision, if applicable (chapter 34.3.1, page 285).

```
163b <SelectBranch 163b>≡ (169b)
    function SelectBranch {
        # Called by BuildApp TaskSelect BuildNewRevision

        CheckGitStatus
        DebianBranches
    }
    <SelectBranch1 167>
```

After the *CheckGitStatus* function, described in the following section, the *DebianBranches* function is called to determine the local Debian branches in the Git repository (chapter 30.4.3, page 166).

30.4.1. Check with *git status*.

The *CheckGitStatus* function, which is also called elsewhere in the program script, checks whether there are new or changed files in the current Git branch. If this is the case, some Git operations cannot be performed.

If there are no problems, we continue with the display of the existing Git branches (chapter 30.4.4, page 167).

If this is detected by *git status*, the *FailureNotice* function is called and a possibility for error recovery in another terminal is opened (chapter 30.4.2, page 165).

```

164  <CheckGitStatus 164>≡ (207b)
      function CheckGitStatus {
          # Called by Import2Git BuildWithUscan PQImport
          # PatchesTreatment SelectBranch and itself

          # Checks git status
          echo "Notice from CheckGitStatus:" >> ${log}
          echo "$(git status) >> ${log}

          if [ ! -z "$(git status --short)" ]
          then
              git status
              FailureNotice "'git status' shows problems\n\
              Please clean up 'git status'"
              if whiptail --title "Check another time?" \
              --yesno "Do you want to check the git status another time?" \
              --yes-button "Yes" --no-button "No" 15 60
              then
                  CheckGitStatus
              fi
          fi
      }

      <CheckTags 216>

```




Figure 30.3.: Query - Further check?

30.4.2. Error message and troubleshooting

If the *FailureNotice* function is called, a message is displayed on the terminal and the possibility is opened to correct errors in another terminal.

This function can be given a special text as a parameter when it is called. If this is not done, a standard text is output.

```

165 <FailureNotice 165>≡ (151)
    function FailureNotice {
        # Called by PQImport CheckGitStatus RebasePQBranch PQMigration
        # You can call this function with a text as parameter (optional)

        if [ ! "$1" ]
        then
            echo "Failure" echo "Something went wrong!"
        else
            for i in $*
            do
                String=${String}" "$i
            done

            echo -e ${String}

        fi
        echo
        echo "Break for fixing it in another terminal"
        echo "After fixing press RETURN to go on!"
        read a
    }

<PQImport 177>

```

30.4.3. Selection of the Debian branches

The *DebianBranches* function determines which relevant branches exist in the Git repository.

To do this, the *IdentifyBranches* function is called first (chapter 29.5.2, page 152).

```
166a  <DebianBranches 166a>≡ (168)
      function DebianBranches {
        # Called by CreateNewBranch SelectBranch
        # selects the Debian branches
        IdentifyBranches
```

```
<DebianBranches1 166b>
```

In the following, the selection is limited to the local Debian branches.

```
166b  <DebianBranches1 166b>≡ (166a)
      ## Trim branchlist
      bl=$(echo $bl | sed 's/pristine-tar/ /')
      bl=$(echo $bl | sed 's/upstream/ /')
      bl=$(echo $bl | sed 's/HEAD/ /')
      # bl=$(echo $bl | sed 's/remotes\origin\./ /g')
      # bl=$(echo $bl | sed 's/remotes\salsa\./ /g')
      # bl=$(echo $bl | sed 's/remotes\home\./ /g')
      bl=$(echo $bl | sed 's/remotes\./ /g')
  }
```

```
<CreateNewBranch (never defined)>
```

30.4.4. Dialog to select a branch

This dialog is only displayed if there are multiple Debian branches. The current branch is preselected.

If only one or no branch exists, we continue in chapter 30.4.7 (page 170).

```

167  <SelectBranch1 167>≡ (163b)
    ## Create a radiolist with the branch names
    ba=($bl) i=1; slct=''
    for element in ${ba[*]}
    do
        echo ${element} | grep 'x_' > /dev/null
        if [ $? -eq 0 ]
        then
            if [ "${element}" = "x_" ]
            then
                continue
            else
                ostr="on"
            fi
        else
            ostr="off"
        fi
        slct=${slct}' '$i' '$element}' '$ostr}' '
        i=$((expr $i + 1))
    done

    if [ ${#ba[@]} -gt 1 ]
    then
        ## select branch
        branch=$(whiptail --title "Branch" --radiolist "Select:" \
            15 60 8 ${slct} --cancel-button "Cancel" 3>&2 2>&1 1>&3)
        if [ ! -z "${branch}" ]
        then
            branch=$((expr ${branch} - 1))
            bName=${ba[$branch]}
            bName=$(echo ${bName} | sed --expression='s/^x_//')
            ## checkout branch
            git checkout ${bName}
            ## Change config file -
            ## make selected branch to recent one
            ChangeEntry
        <SelectBranch3 169a>

```

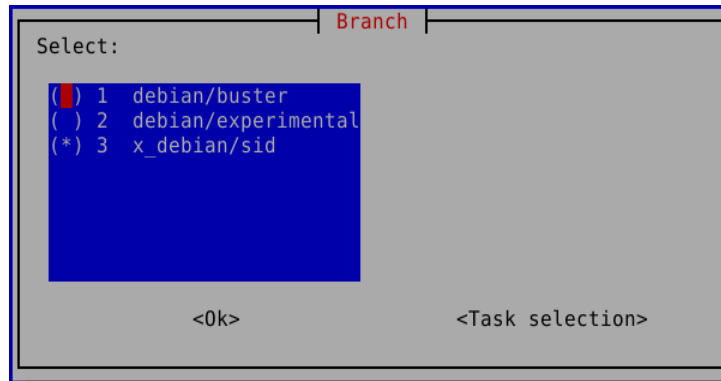


Figure 30.4.: Selection of the Debian branch.

Instead of *debian-stable* the name chosen for this Git branch is displayed.

If this is not the case, the program can be exited by clicking on the *Cancel* button and in the next step with *Exit*.

Now the configuration file is displayed again with *less*.

30.4.5. Change entry

```
168  <ChangeEntry 168>≡
    function ChangeEntry {
        # Called by CreateNewBranch SelectBranch

        RecentBranchEntry=$(grep --count 'RecentBranch=' ${ConfigPath}${OrigName})

        ## Change RecentBranch entry in config file
        if [ ${RecentBranchEntry} -eq 0 ]
        then
            echo "RecentBranch=${bName} >> ${ConfigPath}${OrigName}"
        else
            # ReplaceConfigLines needs two parameters:
            # name of the variable and new value
            ReplaceConfigLines 'RecentBranch' ${bName}
            # bName1=$(echo ${bName} | sed --expression='s/\//\\//g')
            # sed --in-place --expression=\ # "s/RecentBranch=./RecentBranch=${bName1}/g" \
            # ${ConfigPath}${OrigName}
        fi
        less --LINE-NUMBERS ${ConfigPath}${OrigName}

        ## Set variable
        RecentBranch=${bName}
        echo "Notice from ChangeEntry: The branch is "${RecentBranch} >> ${log}

    }

    <DebianBranches 166a>
```

```

169a  <SelectBranch3 169a>≡
      whiptail --title "This branch was selected" \
      --msgbox "${bName} was selected" 15 60
      echo "${bName} was selected" >> ${log}
      ParseConfig
    fi
  <SelectBranch4 170a>

```

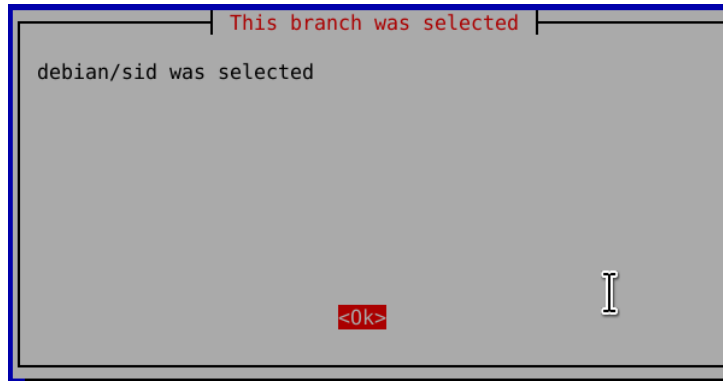


Figure 30.5.: Selected Debian branch.

In the user interface it continues with the task selection (chapter 30.5, page 171).

30.4.6. Read configuration

This function maps the Debian distribution for which the package is built to the selected Git branch. (Chapter ??, page ??)

```

169b  <ParseConfig 169b>≡
      function ParseConfig {
        # Called by SelectBranch TaskSelect

        # Parse config file for Debian distribution of branch
        vc=$(grep --count ${bName}_Dist ${ConfigPath}${OrigName})
        if [ $vc -ge 1 ]
        then
          Search4Dist
        else
          va="sid"
        fi
        RecentBranchD=${va}
        echo "Notice from ParseConfig: The distribution is "${RecentBranchD} >> ${log}
      }

  <SelectBranch 163b>

```

30.4.7. No or only one branch exists

```
170a  <SelectBranch4 170a>≡ (169a)
      elif [ ${#ba[@]} -eq 1 ]
      then
        whiptail --title "Only one branch" \
          --msgbox "There is only one Debian branch: ${ba[0]}" 15 60
      <SelectBranch6 170b>
```

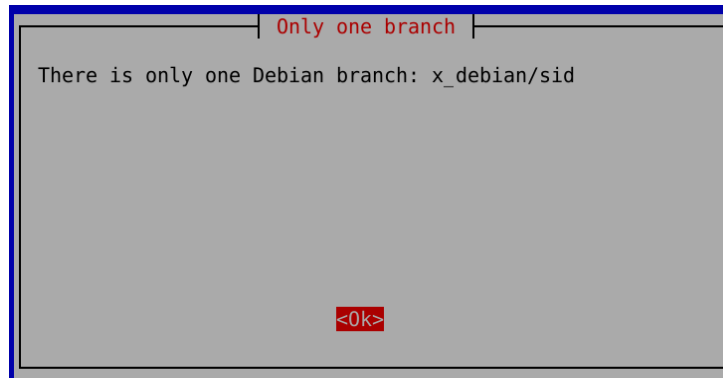


Figure 30.6.: There is only one Git branch

If there is a Git branch, the task selection continues (chapter 30.5, page 171). Otherwise there is a hint.

```
170b  <SelectBranch6 170b>≡ (170a)
      else
        whiptail --title "There is no branch" \
          --msgbox "There is no branch created.\nPlease build a new version." 15 60
      fi
    }

    <AddGitServer (never defined)>
```

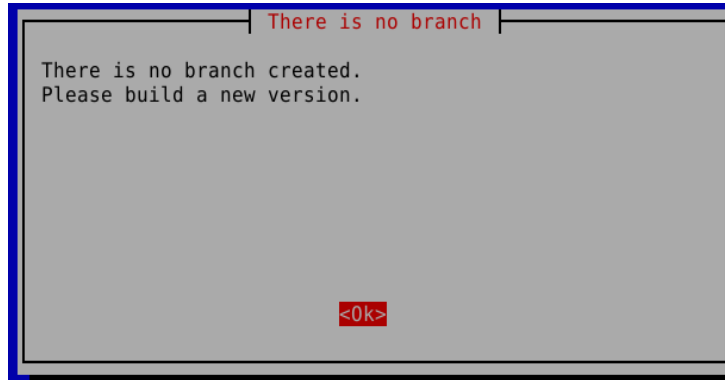


Figure 30.7.: No branch created

30.5. Task selection

After the configuration file is loaded and checked or created, a task selection menu appears.

Since the program is modular, the tasks can also be selected individually. The building process can be interrupted and resumed later.

```

171  <CommonTasks 171>≡ (156b)
      function CommonTasks {
          # Called by BuildApp TaskSelect

          Task=$(whiptail --title "Tasks:" \
            --radiolist "What do you like to do? " 17 60 9 \
              "2" "Build a new version of a package" off \
              "3" "Build a new debian revision" on \
              "4" "Rebuilding a revision" off \
              "5" "Running lintian and uscan" off \
              "6" "Uploading only (build again if necessary)" off \
              "7" "Create new branch" off \
              "8" "Select branch" off \
              "9" "Set name or IP of own git server" off \
              10" "Create a debdiff" off \
            --cancel-button "Exit" 3>&2 2>&1 1>&3)

          if [ -z "${Task}" ]
          then
              exit
          fi
          TaskSelect
      }

      <AskOrigName 106a>

```

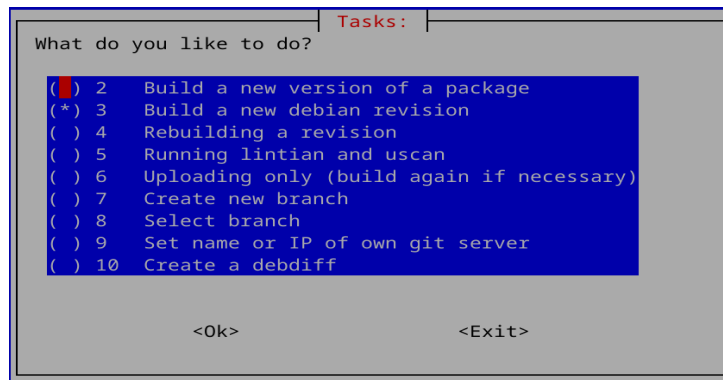


Figure 30.8.: Task selection.

The choices are first building a new version (chapter 31, page 175) and a new revision. As *default* it continues with building a new Debian revision (chapter 32, page 225).

If a new version is to be built, it asks whether changes should be downloaded from *salsa* first (chapter 31.1, page 175). After that, the program script checks if there are already patches. In such a case, these patches can be imported into a separate Git branch as *patch-queue* (see chapter 31.2, page 177).

After building, the packages can be checked with (chapter 37, page 311).

The next menu item concerns uploading (chapter 38, page 325).

Two further tasks concern the case of working with multiple Git branches, for example also for backports (chapter 30.4, page 163). Finally, you can also include your own Git server in the workflow (chapter 41.2, page 356).

The following lines of the script are also used for flow control. A called function changes the variable 'Task' at its end so that one of the following if-clauses applies. For this reason, the following lines cannot be replaced by a case statement.

The chain of If statements gives the possibility to process one condition after the other, because the called function returns to the end of the If statement from which it was called. A case statement ends with the call of the function to be processed.

The call of the corresponding functions is done in the mentioned way again by the function *TaskSelect*.

The following call invokes the build of a new version (chapter 31.1, page 175). there, it first checks if any changes or additions already exist in the project repository on *salsa.debian.org*.

172 <TaskSelect3 172>≡ (135)

```
## Common tasks
rcts=0 # ReCall TaskSelect flag

# Building a new version
if [ $Task -eq 2 ]
then
    PullFromSalsa # and then downlod new version
fi
```

<TaskSelect4 173a>

Or it continues with building a new Debian revision (chapter 32, page 225). This corresponds to the preset value.

```
173a  <TaskSelect4 173a>≡ (172)
      # Building a new revision
      if [ $Task -eq 3 ]
      then
          BuildNewRevision
      fi
```

```
<u1>203<u2>NW4L0rMe-11Dd8c-1<u3><u4><u6>NW4L0rMe-11Dd8c-1<u8>TaskSelect4~<u5><u7>NW4L0rMe-11Dd8c-1<u9>
      # Building a revision
      if [ $Task -eq 3 ]
      then
          BuildNewRevision
      fi
```

<TaskSelect5 173b>

After building a new revision, test it. (Chapter 37, page 311) further. This is done using *lintian* (chapter 37.3.1, page 314) and *uscan* (chapter 37.4, page 316).

```
173b  <TaskSelect5 173b>≡ (173a)
      # Running lintian and uscan
      if [ $Task -eq 5 ]
      then
          RunningTests
      fi
      <TaskSelect6 173c>
```

The next step calls the *PrepareUploading* function (chapter 38, page 325). In it, it is first checked whether the file *debian/changelog* is already prepared for a release . If not, this is still done.

```
173c  <TaskSelect6 173c>≡ (173b)
      # Uploading the package
      if [ $Task -eq 6 ]
      then
          PrepareUploading
      fi
```

<TaskSelect7 174>

The following section is necessary to build for a different distribution. This refers to Debian backports, Debian Proposed updates for stable and possibly for oldstable. If necessary, this is also needed to provide a package for an Ubuntu branch. (See also chapter 36, page 309).

```

174  <TaskSelect7 174>≡ (173c)
      # Create new branch
      # (e.g. for backports or proposed-updates)
      if [ $Task -eq 7 ]
      then
          CreateNewBranch
          rcts=1
      fi

      # Select branch
      if [ $Task -eq 7 ]
      then
          SelectBranch
          rcts=1
      fi

      # Set name or IP of own git server
      if [ $Task -eq 9 ]
      then
          OwnServer
          rcts=1
      fi

      # Create a debdiff
      if [ $Task -eq 10 ]
      then
          DebDiff 0
          rcts=1
      fi

      <TaskSelect9 (never defined)>

```

31. Building a new version

This chapter describes the steps leading to the creation of the **Debian** source code package (**.orig.tar.(g/x/z)*) (Chapter 8, page 23). To do this, also start the program script (chapter 28.2, page 104).

The program script makes it possible to obtain the upstream source code in various ways. Two possibilities have already been described, namely cloning *salsa.debian.org* (Chapter 29.5, page 149) and importing a **Debian** source code package (Chapter 29.6, page 155).

[fuzzy]How a shell script helps to build a *Debian* package

31.1. Download changes from *Salsa*.

It can happen, especially with team-supported packages, that other team members provide changes in the Git repository on *salsa.debian.org*. For further work on this repository it is now mandatory to commit these changes to the local repository as well.

This can be done with

```
gbp pull --all salsa
```

If this causes problems, you can also update the individual branches with

```
git checkout <BranchName>
git pull salsa <BranchName>.
```

```
175  <PullFromSalsa 175>≡ (184b)
      function PullFromSalsa {
          # Called by TaskSelect

          cd ${GitPath}
          if git remote --verbose | grep --quiet 'salsa'
          then
              if whiptail --title "Pull from Salsa?" \
                  --yesno "Do you like to pull possible changes from salsa?" \
                  --yes-button "Yes" --no-button "No" 15 60
              then
                  echo "RecentBranch: "$(git branch) >> ${log} git pull --all
              fi
          <PullFromSalsa1 176>
```

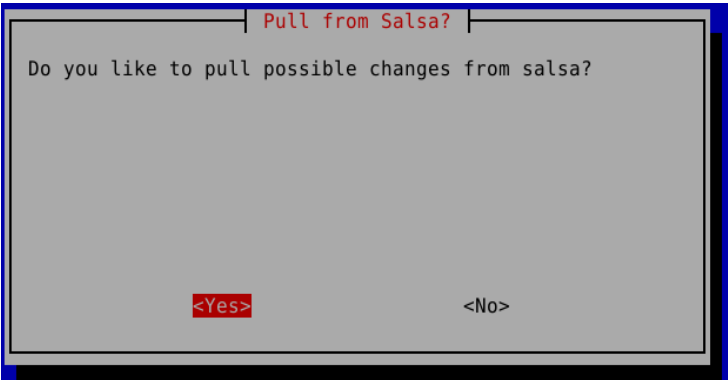


Figure 31.1.: Download from salsa.debian.org

In addition, the password for the SSH key is requested, which can be used to access *salsa.debian.org*.

```
176  <PullFromSalsa1 176>≡ (175)
      fi
      ClassicalOrUscan
    }
    <CloneFromSalsa 149a>
```

The *PullFromSalsa* function calls the *ClassicalOrUscan* function (chapter 31.3, page 182) at the end, which first calls the *PQImport* function to import an existing *patch queue*.

31.2. Import an existing *patch queue*

Before downloading the new upstream version, it checks if there is already a *patch queue* for the previously packed version. By "*patch queue*" is meant the patches listed in the *debian/patches/series* file in the order there.

In this case, the possibility is opened to import them via *gbp pq import* into a separate *patch-queue-branch*, if this does not already exist. This allows the patches to be added back into it later – after the new version has been imported.

When imported into the patch queue branch, the patches are applied to the upstream source code.

This requires that there are no unversioned files in the current *Git* branch and that all patches listed in the *debian/patches/series* file can be applied. Otherwise, *gbp pq import* will fail. Then no *patch-queue branch* is created. Making the patches manually applicable can be time-consuming.

However, the above conditions should usually be met before importing a new version.

```

177 <PQImport 177>≡ (165)
    function PQImport {
        # Called by ClassicalOrUscan PQMigration CloneFromSalsa and itself
        returnflag=$1
        if [ ! ${returnflag} ]
        then
            returnflag=1
        fi
        cd ${GitPath}
        if echo $(git branch) | grep --quiet 'patch-queue/'${RecentBranch}
        # patch-queue branch already exists
        then
            return
        fi

        if [ -f debian/patches/series ]
        # debian/patches/series exists
        then
            if whiptail --title "There are patches" \
            --yesno "Do you like to import the current patches\n\
            onto the patch-queue branch? (recommended)" \
            --yes-button "Yes" --no-button "No" 15 60
            <PQImport1 178a>

```

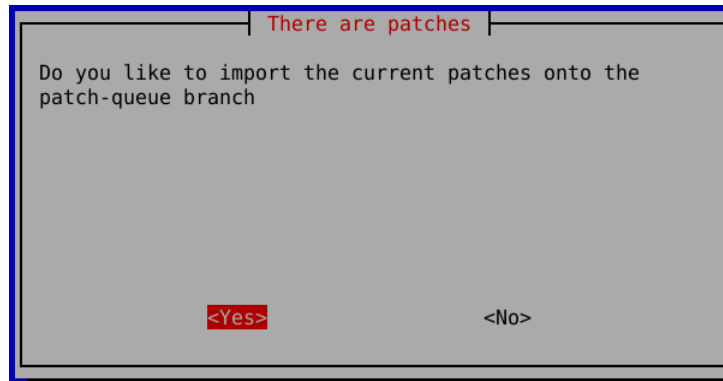


Figure 31.2.: There are patches

If this question is answered in the negative, the download of the source code continues (chapter 31.3, page 182).

Otherwise, in preparation for importing into a *patch queue* branch, the next step is to check the Git status as described in chapter 30.4.1 (page 164).

```
178a  <PQImport1 178a>≡ (177)
      then
        CheckGitStatus
      <PQImport1-1 178b>
```

31.2.1. First attempt to import

If no patch queue branch exists, which should be the normal case, one is created. Then the import into this patch queue branch takes place. After the import has been completed, the original branch (usually *debian/sid*) is (re)switched to (chapter 31.2.3, page 180)..

All patches listed in the *debian/patches/series* file must be applicable.

```
178b  <PQImport1-1 178b>≡ (178a)
      echo "Notice from gbp pq import: " >> ${log}
      gbp pq --verbose import >> ${log} 2>&1
      &
      if [ $? -eq 1 ]
      then
        Notice="All patches listed in debian/patches/series\n\
        have to be applicable"'\''\n\
        For Details, look into the log file of the project.\n"
        FailureNotice ${Notice}
      <PQImport2 179a>
```

If the import fails, a troubleshooting option is opened (chapter 30.4.2, page 165). After that, a new attempt can be made. The successful import is reported by the program script (chapter 31.2.3, page 180).

31.2.2. Another import attempt

After the error has been corrected, a new import attempt can be made. If the error correction attempt is deemed unsuccessful, the import can be canceled.

```
179a  <PQImport2 179a>≡ (178b)
      if whiptail --title "Fixed?" --yesno "Retry?" \
        --yes-button "Yes" --no-button "No import" 15 60
      <PQImport3 179b>
```

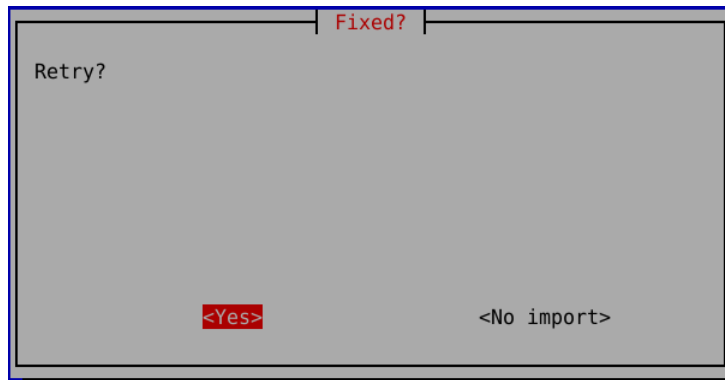


Figure 31.3.: Fixed? Retry?

```
179b  <PQImport3 179b>≡ (179a)
      then
        PQImport ${returnflag}
      else
        whiptail --title "No import onto a patch-queue branch" \
          --msgbox "Let's go on without the import\n\
            of the current patches onto a patch-queue branch" \
            15 60
      fi
      <PQImport4 180>
```

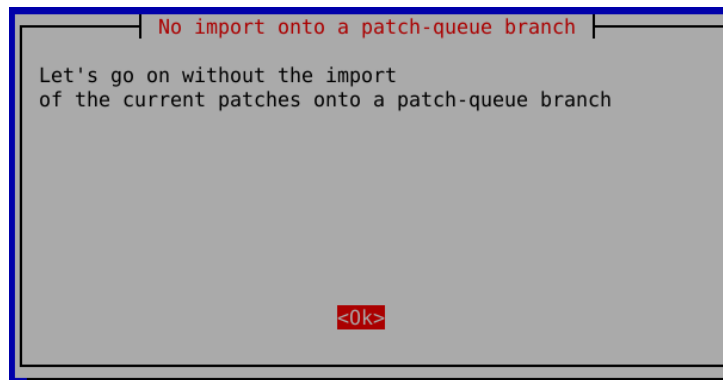


Figure 31.4.: No import into patch queue

31.2.3. Import successful into PQ branch

The successful import into the PQ-Branch is indicated with a dialog.

```

180  ⟨PQImport4 180⟩≡ (179b)
      else
        whiptail --title "Done" \
          --msgbox "Imported the current patches onto the patch-queue branch" \
            15 60
      ⟨PQImport5 181⟩

```

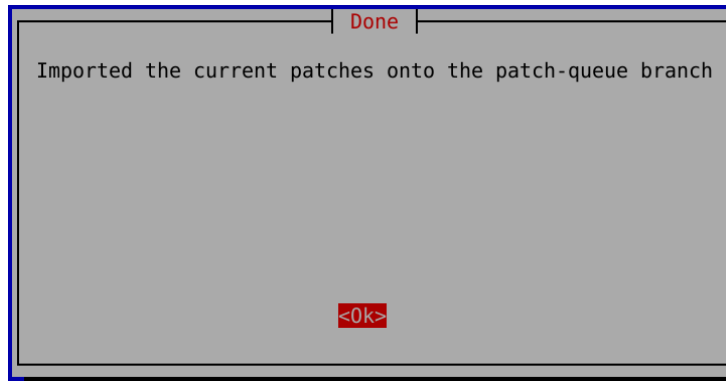



Figure 31.5.: PQ-Import successful

Then the program script returns to the original Git branch (Debian branch).

```

181  ⟨PQImport5 181⟩≡
      if [ ${returnflag} -eq 1 ]
      then
        # Back to the previous branch
        git checkout ${RecentBranch}
      fi
    fi
  fi
}
⟨ClassicalOrUscan 182a⟩

```

(180)

31.3. Tools for downloading the upstream sources.

To build a **Debian** package, the source code of the upstream project is needed first. This can be obtained in several ways.

Downloading the source code with *wget* is the classic method (chapter 31.4, page 185). This way should be chosen when building a new package for the first time.

It happens that upstream does not provide a source code archive. Instead, source code can be found at *Github* or similar hosts. This places special demands on the maintainer of a **Debian** package.

Alone to build *reproducible*, a tar archive of the upstream source code must be provided. This must be possible without network access to External Resources.

This method must also be chosen if a specific state of the upstream code from a Git repository (e.g. Github or similar) is to be used.

However, if a file *debian/watch* and other files already exist in the directory *debian/*, *uscan* (chapter 31.5, page 221) can also be used for this. A download with *uscan* is recommended if the file *debian/watch* contains an entry *uversionmangle=*.

If *uscan* cannot identify the new version, the new version must be deployed manually or via *wget*.

These possibilities are offered alternatively by the program script.

There is also the possibility to use *get-orig-source* in *debian/rules*.

```
182a  <ClassicalOrUscan 182a>≡ (181)
      # Called by PullFromSalsa
```

```
# Before importing a new version, check whether there is a patch-queue, # which can be exported onto
```

```
<ClassicalOrUscan1 182b>
```

The *PQimport* function is described in chapter 31.2 (page 177).

If a *debian/watch* file already exists, the user is asked if he wants to download the new version in the "classical" way or with *uscan*..

```
182b  <ClassicalOrUscan1 182b>≡ (182a)
      if [ ! -f ${GitPath}/debian/watch ]
      then
          BuildNewVersion
          return
      <ClassicalOrUscan1-1 183a>
```

It is checked whether in the file *debian/watch* *addons.thunderbird.net* or *addons.mozilla.org* are listed as source. Because from there no download with uscan is possible.

183a $\langle \text{ClassicalOrUscan1-1 } 183a \rangle \equiv$ (182b)

```
else
  # Download from Mozilla repos is not possible with uscan
  if grep --quiet "addons.thunderbird.net" ${GitPath}/debian/watch
  then
    whiptail --title "Thunderbird Repository" \
      --msgbox "From addons.thunderbird.net \nyou can't dowload with uscan" \
      15 60
    BuildNewVersion
    return
  fi
```

$\langle \text{ClassicalOrUscan2 } 183b \rangle$

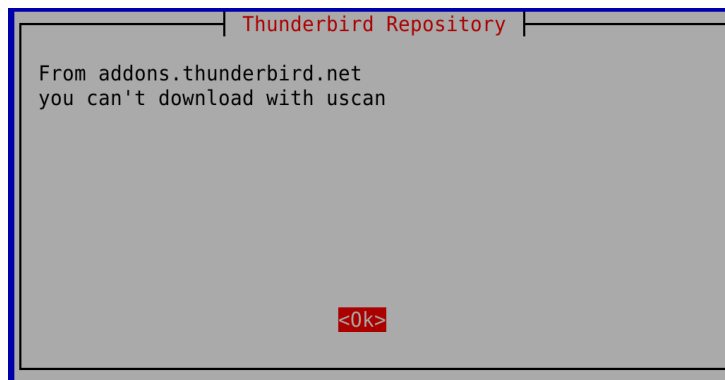


Figure 31.6.: No download via uscan from thunderbird.net

183b $\langle \text{ClassicalOrUscan2 } 183b \rangle \equiv$ (183a)

```
if grep --quiet "addons.mozilla.org" ${GitPath}/debian/watch
then
  whiptail --title "Mozilla Repository" \
    --msgbox "From addons.mozilla.org \nyou can't dowload with uscan" \
    15 60
  BuildNewVersion
  return
fi
```

$\langle \text{ClassicalOrUscan3 } 184a \rangle$

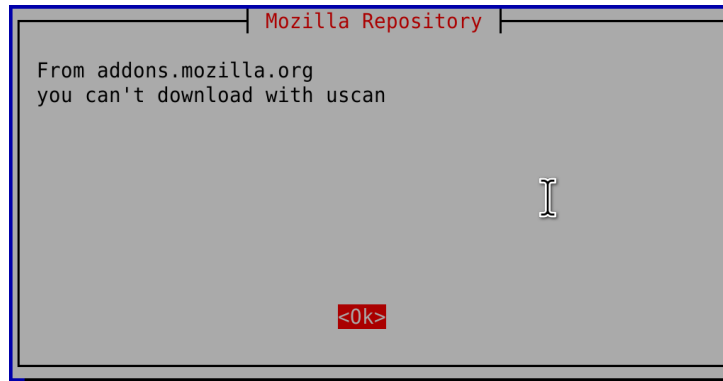


Figure 31.7.: No download via uscan from mozilla.org

184a $\langle \text{ClassicalOrUscan3 } 184a \rangle \equiv$ (183b)
 NVTask=\$(whiptail --title "Classical download or uscan" \
 --radiolist "How do you want to download the new version? " 17 60 9 \
 "0" "using the classical way" on \
 "1" "using uscan" off --cancel-button "Exit" 3>&2 2>&1 1>&3)

$\langle \text{ClassicalOrUscan5 } 184b \rangle$

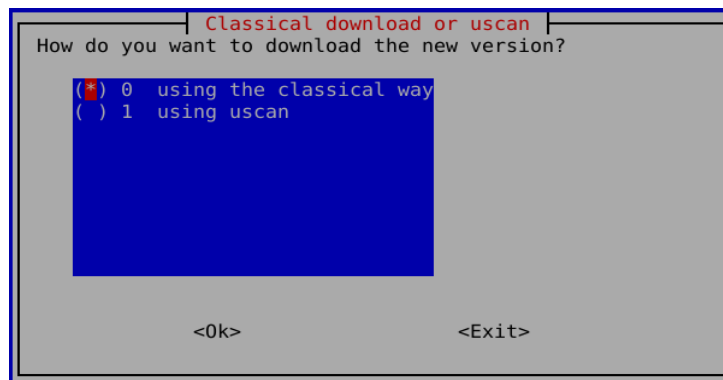


Figure 31.8.: Download - classical or with uscan

184b $\langle \text{ClassicalOrUscan5 } 184b \rangle \equiv$ (184a)
 if [-z "\${NVTask}"]
 then
 exit
 case "\$NVTask" in
 0) BuildNewVersion;;
 1) BuildWithUscan;;
 esac
 fi
 }

$\langle \text{PullFromSalsa } 175 \rangle$

If the *debian/watch* file does not exist, the user must download the new version in the classical way.

If downloading with *uscan* is selected instead of the classical way, it continues with chapter 31.5 (page 221).

31.4. Download the classic way

As a rule, the source code of a software is provided as an archive. Various formats are available for this purpose. The use of these is described below.

31.4.1. Archive formats

For use with *git-buildpackage*, an orig-tar archive is mandatory as source. This may only have the formats **.tar.gz* or **.tar.xz*. The orig tar archive is also uploaded to the Debian archive.

For use on Linux, a **.tar.gz* is usually provided. Sometimes this is also a **.tar.xz*.

For software that spans multiple operating systems, the source code is often provided as a zip archive. A zip archive is therefore repacked into an *.tar.xz*. For this there is the tool *mk-origtargz* (chapter 31.4.6, page 203).

In addition, in the file *debian/gbp.conf* belonging to the project can be specified which archive format should be used. If *compression = xz* is specified here as compression, a **.tar.gz* must also be converted to a **.tar.xz*. This is then documented in the *debian/README.source* file (chapter 32.4.18, page 249)..

Use *mk-origtargz* to rename the original authors' tarball, optionally change the compression, and remove unwanted files.

31.4.2. Downloading the source code

So first the source code is downloaded.

```

185 <BuildNewVersion 185>≡ (220b)
    function BuildNewVersion {
        # Called by ClassicalOrUscan

        echo "Building a new version" >> ${log}

        UpstreamSourceName=$(whiptail --title "Name of the source" \
        --inputbox "Please insert the file name of the upstream source version\n \
        to be downloaded or copied (including version and suffix):\n" \
        --cancel-button "Exit" 15 60 3>&2 2>&1 1>&3)

    <BuildNewVersion3 186a>

```

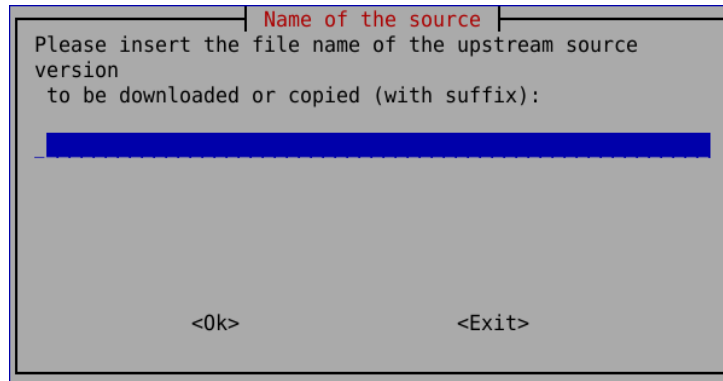


Figure 31.9.: Name of the upstream URL

Now the name of the source code package is entered here.

186a `<BuildNewVersion3 186a>≡` (185)

```

    if [ -z "${UpstreamSourceName}" ]
    then
        exit
    fi

```

`<BuildNewVersion4 186b>`

The program takes care of downloading the upstream version. If this has already been downloaded, the program can also continue to work with it.

186b `<BuildNewVersion4 186b>≡` (186a)

```

    cd ${PrjPath}
    if whiptail --title "Should the source be downloaded?" \
    --yesno "Should $UpstreamSourceName\n \
    be downloaded from the upstream page?" \
    --defaultno --yes-button "Yes" --no-button "No" 15 60
    <BuildNewVersion4-1 187a>

```

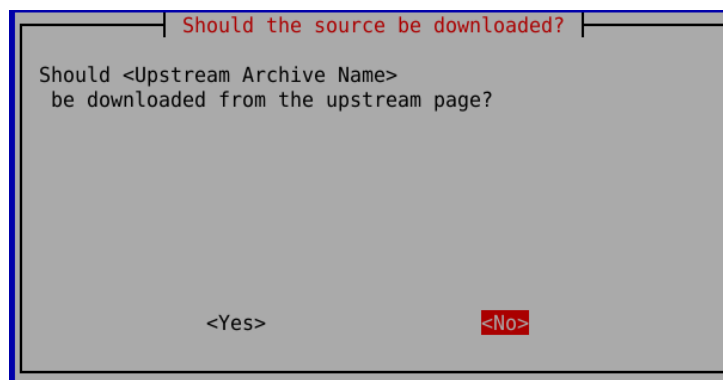


Figure 31.10.: Download (or copy)?

If the question is answered in the negative, the next step is copying (chapter 31.4.2.2, page 189).

31.4.2.1. Download

```

187a  <BuildNewVersion4-1 187a>≡ (186b)
      then
        if [ -z "${DownloadUrl}" ]
        then
          DownloadUrl=$(whiptail --title "Insert URL for download" \
            --inputbox "Please insert the complete\n \
            URL to download ${UpstreamSourceName}\n(with 'https://'\n \
            or so and the name of the archive):" \
            --cancel-button "Exit" 15 60 3>&2 2>&1 1>&3)
        fi

      <BuildNewVersion4-2 187b>

```

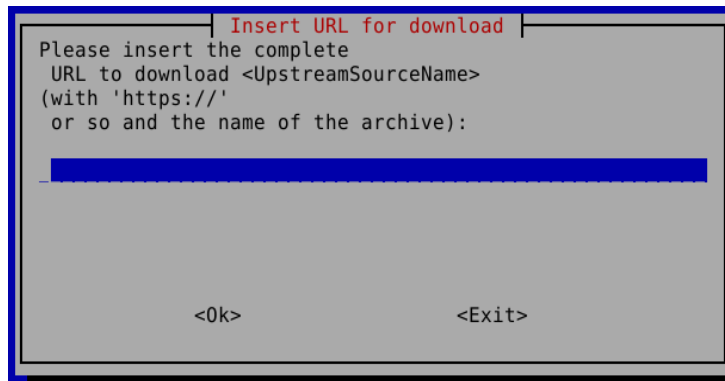


Figure 31.11.: Enter link for download

```

187b  <BuildNewVersion4-2 187b>≡ (187a)
      if [ -z "${DownloadUrl}" ]
      then
        exit
      else
        changeflag=1
      fi

```

<BuildNewVersion5 187c>

As a precaution, you will be asked if the URL to download the source code is correct.

```

187c  <BuildNewVersion5 187c>≡ (187b)
      if ! whiptail --title "DownloadUrl" \
        --yesno "The complete URL to download ${UpstreamSourceName} is\n \
        ${DownloadUrl}" --yes-button "Yes" --no-button "No" 15 60
      <BuildNewVersion5-1 188a>

```

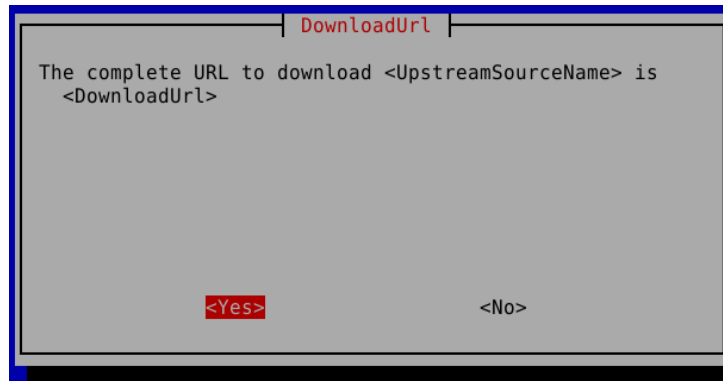


Figure 31.12.: Download-URL right?

```
188a  <BuildNewVersion5-1 188a>≡ (187c)
      then
        DownloadUrl=$(whiptail --title "Complete URL" \
          --inputbox "Real complete URL to download ${UpstreamSourceName}\n \
            (with 'https://' or so and the name of the archive):" \
          --cancel-button "Exit" 15 60 3>&2 2>&1 1>&3)
      fi
```

<BuildNewVersion5-2 188b>

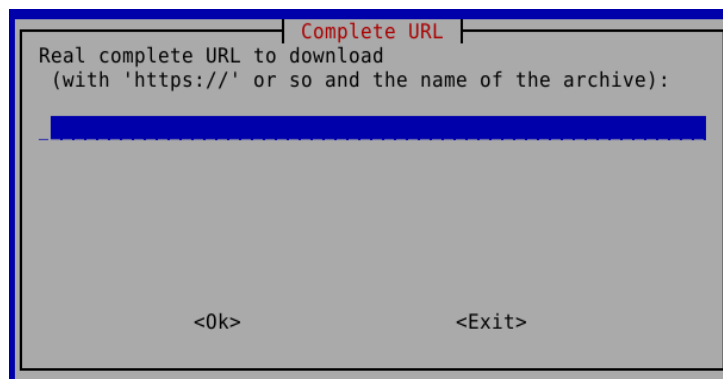


Figure 31.13.: Correct download URL

```
188b  <BuildNewVersion5-2 188b>≡ (188a)
      if [ -z "${DownloadUrl}" ]
      then
        exit
      else
        changeflag=1
      fi
      <BuildNewVersion6 189a>
```


The new URL for downloading the source code is entered in the configuration file. After that the download is done by *wget*.

Then the possibility is opened to download and check also the signature file (chapter 31.4.7, page 206).

```
189a  <BuildNewVersion6 189a>≡ (188b)
      # Write download URL into config file
      if [ $changeflag -eq 1 ]
      then
          ReplaceConfigLines 'DownloadUrl' ${DownloadUrl}
          changeflag=0
      fi

      # getting sources using wget
      wget --verbose $DownloadUrl &&
      echo -e "The sources were pulled from\n${DownloadUrl}\n \
      by wget." >> ${log}

      if whiptail --title ".asc file?" \
      --yesno "Do you want to download an .asc file, too?" \
      --yes-button "Yes" --no-button "No" 15 60
      then
          DownloadAscFile
      fi
      <BuildNewVersion6-1 189b>
```

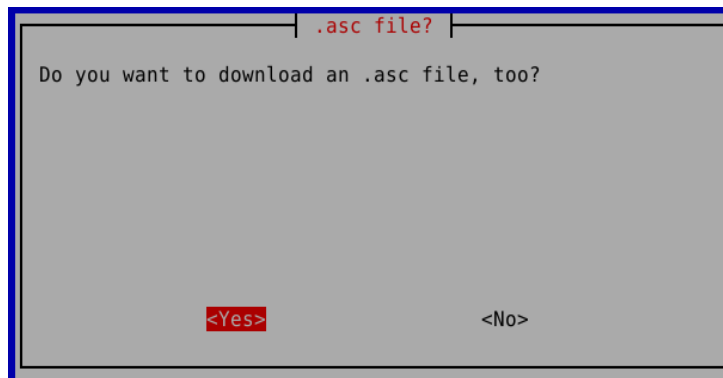


Figure 31.14.: Download *.asc file

31.4.2.2. Copy the source archive

```
189b  <BuildNewVersion6-1 189b>≡ (189a)
      else
          whiptail --title "Please copy the source code now" \
          --msgbox "Please copy ${UpstreamSourceName} to ${PrjPath}!" 15 60
      <BuildNewVersion6-2 190a>
```



Figure 31.15.: Path to copy

```

190a  <BuildNewVersion6-2 190a>≡                                     (189b)
      if ! whiptail --title "Copy finished?" \
        --yesno "Was ${UpstreamSourceName} copied to ${PrjPath}?" \
        --yes-button "Yes" --no-button "No" 15 60
      <BuildNewVersion6-3 190b>

190b  <BuildNewVersion6-3 190b>≡                                     (190a)
      then
        echo "Exit" >> ${log} whiptail --title "Bye" --msgbox "Bye" 15 60
        exit
      fi
      <BuildNewVersion6-4 190c>

```

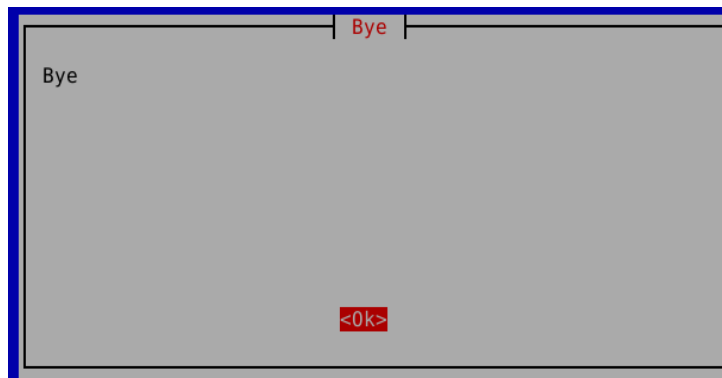


Figure 31.16.: Quit program

```

190c  <BuildNewVersion6-4 190c>≡                                     (190b)
      fi

      # Identify the type of the upstream archive by suffix
      CutSuffix

      <BuildNewVersion7 192a>

```

Now the compression and the version number is determined. This is called also altogether *Suffix*. If everything is in order here it continues in the chapter 31.4.5 (page 195).

31.4.3. Identify suffix

Which compression *mk-origtargz* automatically chooses depends on the file type of the upstream archive. The file name extension is compared with a list of reasonable file types.

191a $\langle CutSuffix \text{ 191a} \rangle \equiv$ (154b)

```
function CutSuffix {
    # Called by BuildNewVersion

    # List of reasonable suffixes
    typea=( '.tar.gz' '.tar.xz' '.tgz' '.zip' '.oxz' '.xpi' '.jar' )
```

$\langle CutSuffix1 \text{ 191b} \rangle$

The file types *.oxz*, *.xpi* and *.jar* all describe *.zip* archives. For the *.xpi* file type, the *mozilla-devscripts* package is required.

191b $\langle CutSuffix1 \text{ 191b} \rangle \equiv$ (191a)

```
KnownTyp=0
for element in ${typea[*]}
do
    if echo ${UpstreamSourceName} | grep ${element} > /dev/null
    then
        echo "Notice from CutSuffix: The suffix of the upstream \
file is "${element}" >> ${log}
        UpstreamSuffix=${element}
        RecentUpstreamSuffix=$(echo ${UpstreamSuffix} | sed --expression s/^\.//)
        ReplaceConfigLines 'RecentUpstreamSuffix' ${UpstreamSuffix}
        KnownTyp=1
    fi
done

if [ ${KnownTyp} -ne 1 ]
then
    echo "Notice from CutSuffix: Unknown suffix" >> ${log}
    if ! whiptail --title "Unknown suffix" \
--yesno "The suffix of ${UpstreamSourceName} is not listed.\n \
Continue anyway?" --defaultno --yes-button "Yes" \
--no-button "No" 15 60
    then
        exit
    fi
fi
}
```

$\langle Name2Version \text{ 192b} \rangle$

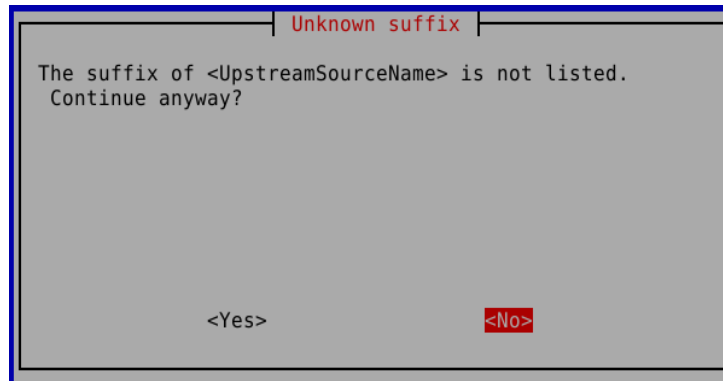


Figure 31.17.: Unknown suffix

31.4.4. Detect upstream version

```

192a  <BuildNewVersion7 192a>≡ (190c)
      # Identify the upstream version number
      Name2Version

      <BuildNewVersion8 192c>

192b  <Name2Version 192b>≡ (191b)
      function Name2Version {
        # Called by BuildNewVersion
        # Extracts version from upstream archive name
        Suffix=$(echo ${UpstreamSuffix} | sed --expression='s/\./\\./g')
        Version1=$(echo ${UpstreamSourceName} | sed --expression="s/${Suffix}$//" | \
        sed --expression="s/.*${SourceName}//gI" | \
        sed --expression='s/-//' | sed --expression='s/v//')
        if [ -z ${Version1} ]
        then
          Version1="0.0.0" # Default value
        fi
      }

      <GbpConfIntegration 208c>

192c  <BuildNewVersion8 192c>≡ (192a)
      if ! whiptail --title "Version" \
      --yesno "You want to build version ${Version1}." \
      --yes-button "Yes" --no-button "No" 15 60
      <BuildNewVersion9 193a>

```

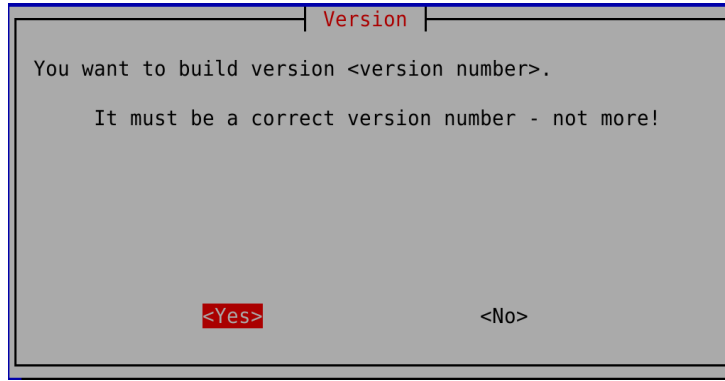


Figure 31.18.: Right version?

It can happen that the program script cannot determine the correct version designation. In this case, the version designation can be entered in the following dialog box, which can also be processed further. In addition to the digits, the version name may only contain dots and the specification whether it is a *release candidate*, a *beta* or *alpha* version or a specific *commit* from the Git repository.

193a `<BuildNewVersion9 193a>≡` (192c)
`then`

```
Version1=$(whiptail --title "Version" \
--inputbox "Name of the upstream version: ${UpstreamSourceName}\n \
Which version (without repack identifiers and without revision)\n \
of the package ${SourceName} should be built?" \
--cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
```

`<BuildNewVersion10 193b>`

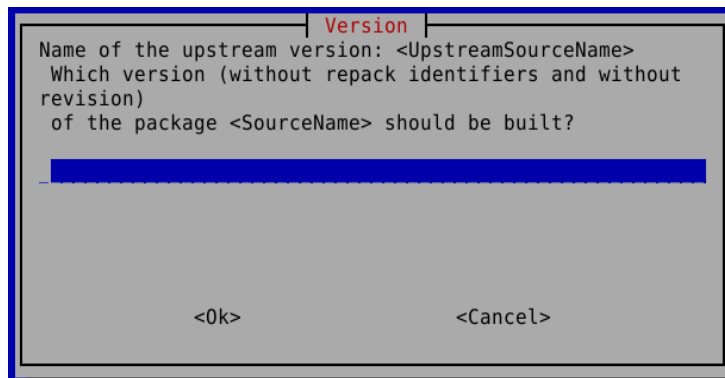


Figure 31.19.: Which version should be built?

193b `<BuildNewVersion10 193b>≡` (193a)

```
if ! whiptail --title "Version" \
--yesno "Do you really want to build version ${Version1}." \
--yes-button "Yes" --no-button "No" 15 60 <BuildNewVersion11 194a>
```

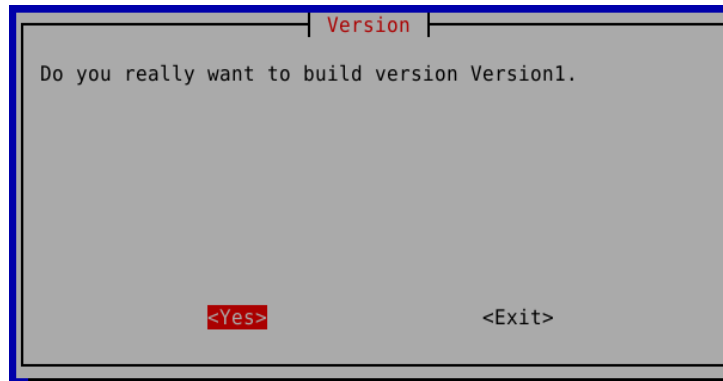


Figure 31.20.: Will correct version be built?

194a $\langle \text{BuildNewVersion11 } 194a \rangle \equiv$ (193b)
 then
 echo "Exit" >> \${log}
 whiptail --title "Bye" --msgbox "Bye" 15 60
 exit
 fi
 $\langle \text{BuildNewVersion11-1 } 194b \rangle$

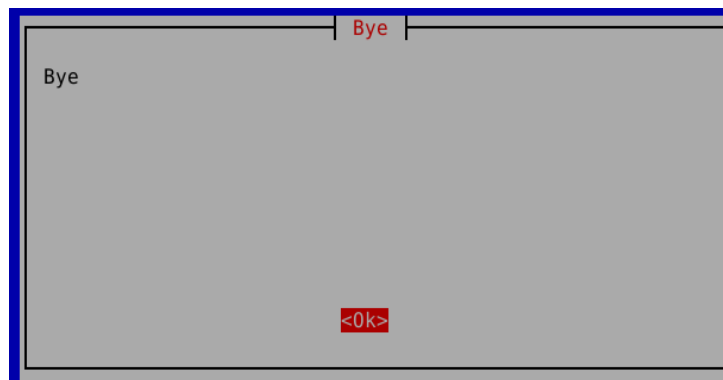


Figure 31.21.: Quit program

194b $\langle \text{BuildNewVersion11-1 } 194b \rangle \equiv$ (194a)
 fi

 ExcludeFiles

 echo "Version "\${Version1}\${ESuffix}" of the package "\${PackName}" \
 should be built." >> \${log}

 $\langle \text{BuildNewVersion12 } 203c \rangle$

31.4.5. Exclude files from upstream archive

Before *mk-origtargz* is called, the program script allows individual source code files to be excluded from inclusion in the *orig* archive.

```
195a  <ExcludeFiles 195a>≡ (199a)
      function ExcludeFiles {
          # Called by BuildNewVersion

      <ExcludeFiles1 195b>
```

It is necessary to specify that files are to be excluded. To do this, the program script determines where the information to exclude files should come from. This is here the file *debian/copyright*.

Now it checks if a file *debian/copyright* exists.

```
195b  <ExcludeFiles1 195b>≡ (195a)
      # Checks whether debian/copyright contains a section Files-Excluded
      gitflag=0
      exflag=0
      crflag=0
      if [ -f ${GitPath}/debian/copyright ]
      then
      <ExcludeFiles2 195c>
```

Then it is checked whether it contains the expression *Files-Excluded*. In this case it is queried whether the file *debian/copyright* should be edited.

```
195c  <ExcludeFiles2 195c>≡ (195b)
      crflag=1
      grep 'Files-Excluded' ${GitPath}/debian/copyright > /dev/null
      if [ $? -eq 0 ]
      then
          exflag=1 whiptail --title "Copyright file contains Files-Excluded" \
          --msgbox "debian/copyright contains section Files-Excluded." 15 60
          less --LINE-NUMBERS ${GitPath}/debian/copyright
      fi
  fi
```

```
<ExcludeFiles3 196>
```

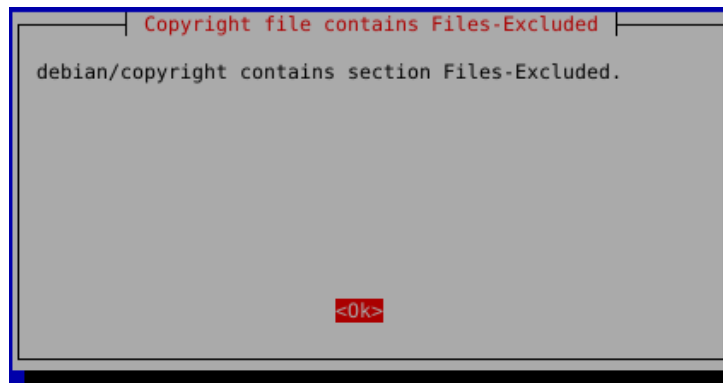


Figure 31.22.: File debian/copyright contains section Files-Excluded

```
196  <ExcludeFiles3 196>≡ (195c)
      if whiptail --title "Exclude files from upstream source" \
      --yesno "Do you want to exclude files from upstream source?" \
      --defaultno --yes-button "Yes" --no-button "No" 15 60
      <ExcludeFiles3-1 197>
```

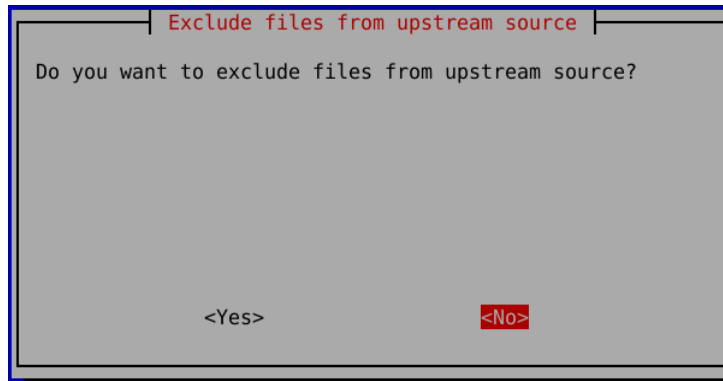



Figure 31.23.: Exclude files

If no file exclusion is needed, *mk-origtargz* is run in the background. (Chapter 31.4.6, page 203). Then it continues in chapter 31.4.8, (page 207).

```

197  <ExcludeFiles3-1 197>≡ (196)
      then
        if [ $crflag -eq 1 ]
        then
          if whiptail --title "Copyright file existst" \
            --yesno "debian/copyright exists.\nDo you want to edit it?" \
            --yes-button "Yes" --no-button "No" 15 60
          then
            gitflag=1
            nano --linenumbers --mouse --softwrap ${GitPath}/debian/copyright
          fi
          AddOpt=" --copyright-file "${SourceName}"/debian/copyright"

```

<ExcludeFiles4 198>



Figure 31.24.: Should debian/copyright be edited?

Otherwise, the *SpecialExcludeFile* function is called. In this function, unless - as in the present case - the information in the file *debian/copyright* is used, a special file is requested which contains the names of the files to be excluded in the format DEP-5¹.

```
198  <ExcludeFiles4 198>≡ (197)
      else
        SpecialExcludeFile
      fi

<ExcludeFiles5 199b>
```

¹DEP-5[17]

If it is already clear before submitting the package to the **New Queue** that files are to be excluded, no *debian/copyright* file exists at that time. It is then a good idea to list the files to be excluded in a separate file.

The *ExcludeFiles* function calls the following function for this purpose. This function asks for a special file containing the names of the files to be excluded in DEP-5 format.

199a $\langle \textit{SpecialExcludeFile}$ 199a \equiv (205b)

```
function SpecialExcludeFile {
    # Called by ExcludeFiles
    if [ -z "${ExcludeFile}" ]
    then
        ExcludeFile=$(whiptail --title "Name of exclude file" \
            --inputbox "Please insert name of the exclude file:" \
            --cancel-button "Exit" 15 60 3>&2 2>&1 1>&3)
        if [ -z "${ExcludeFile}" ]
        then
            exit
        fi
        echo 'ExcludeFile='${ExcludeFile} >> ${ConfigPath}${OrigName}
    else
        whiptail --title "Exclude file name" \
            --msgbox "The name of the exlude file is ${ExcludeFile}" \
            15 60
    fi
    AddOpt=" --copyright-file "${ExcludeFile}
} }
```

$\langle \textit{ExcludeFiles}$ 195a)

Then the program script asks for the attachment to add to the upstream version name..

Often select *+dfsg* as an attachment here to document the reason for the exclusion (see also chapter 10.4.1.3, page 32)

199b $\langle \textit{ExcludeFiles5}$ 199b \equiv (198)

```
ESuffixN=$(whiptail --title "Suffix:" \
    --radiolist "Please choose the suffix: " \
    --cancel-button "Cancel" 15 60 4 \
    "0" "+ds" off \
    "1" "+dfsg" on \
    "2" "other" off 3>&2 2>&1 1>&3)
if [ ${ESuffixN} -eq 1 ]
then
    ESuffix="+dfsg"
elif [ ${ESuffixN} -eq 0 ]
then
    ESuffix="+ds"
```

$\langle \textit{ExcludeFile6}$ 200)

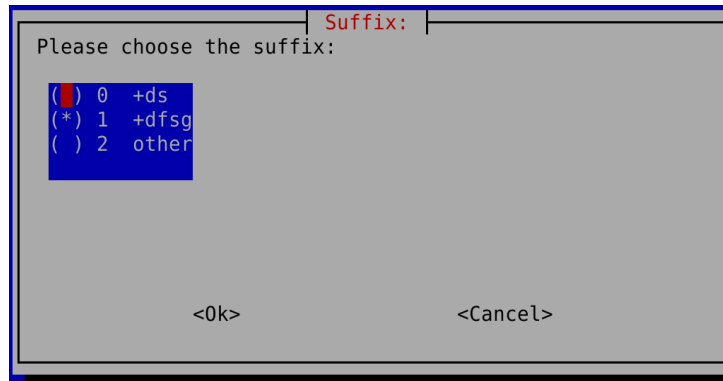


Figure 31.25.: Suffix for exclusion of files

If a suffix other than the two suggested is to be used, this must be added manually. This entry must also be added accordingly in the *debian/watch* file. (s.a chapter 32.4.7, page 238)

```
200  <ExcludeFile6 200>≡ (199b)
      else
        ESuffix=$(whiptail --title "Enter suffix" \
          --inputbox "Please insert the suffix:" \
          --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
      fi
    <ExcludeFile7 201>
```

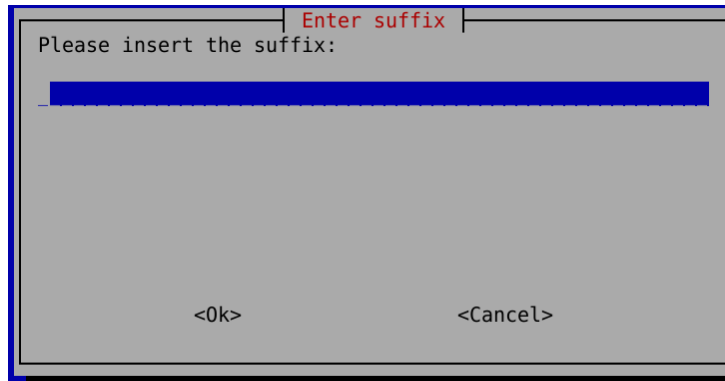


Figure 31.26.: Custom suffix for excluding files

```

201  <ExcludeFile7 201>≡
      if [ -z "${ESuffix}" ]
      then
        whiptail --title "Warning!" \
          --msgbox "You repacked the upstream source and\n\
          do not want to use a repack suffix." 15 60
        if [ -n "${RecentRepackSuffix}" ]
        then
          # Remove suffix from config file
          sed --in-place \
            --expression="s/RecentRepackSuffix=.*//g" \
            ${ConfigPath}${OrigName}
        fi
      else
        <ExcludeFile8 202a>

```

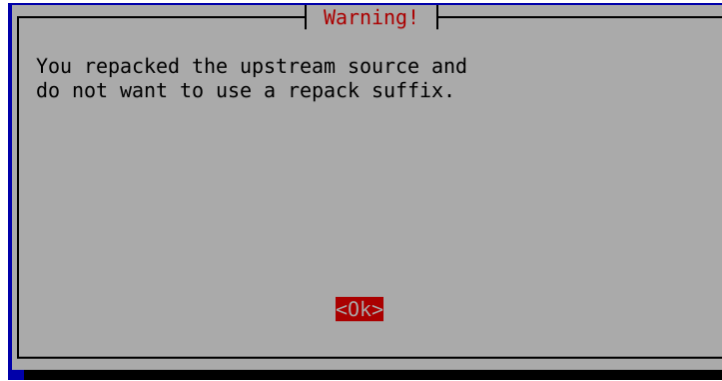


Figure 31.27.: Warning. - No suffix specified

In some cases a plus sign (+) can cause problems especially when building Java packages. Sometimes it is also necessary that no suffix can be added. The documentation of such operations can then be done in the file *README.source* (chapter 32.4.18, page 249)

202a $\langle \text{ExcludeFile8 } 202a \rangle \equiv$ (201)

```
# Insert suffix into config file
if [ -z "${RecentRepackSuffix}" ]
then
    echo "RecentRepackSuffix=${ESuffix} >> ${ConfigPath}${OrigName}
else
    sed --in-place \
        --expression="s/RecentRepackSuffix=.*\/RecentRepackSuffix=${ESuffix}/g" \ ${ConfigPat.
fi

RecentRepackSuffix=${ESuffix}
AddOpt=${AddOpt}" --repack-suffix "${ESuffix}

fi
```

$\langle \text{ExcludeFiles10 } 202b \rangle$

The compression of the **.orig.tar.** archive is stored in the *suffix* variable. It is set to *.tar.xz*.

202b $\langle \text{ExcludeFiles10 } 202b \rangle \equiv$ (202a)

```
whiptail --title "Option(s) for mk-origtargz:" \
    --msgbox "\n${AddOpt}" 15 60
else
    AddOpt=""
    ESuffix=""
 $\langle \text{ExcludeFiles12 } 203a \rangle$ 
```

If no files are to be excluded, but the *debian/copyright* file contains a *Files-Excluded* section, remove it (manually).

```
203a <ExcludeFiles12 203a>≡ (202b)
    if [ $exflag -eq 1 ]
    then
        gitflag=1
        whiptail --title "Copyright file contains Files-Excluded" \
        --msgbox "debian/copyright contains Files-Excluded section.\n \
        Please delete it" 15 60
        nano --linenumbers --mouse \
        --softwrap ${GitPath}/debian/copyright
    fi
fi
```

<ExcludeFiles15 203b>

If the *debian/copyright* file was edited in connection with the exclusion of files, a corresponding *commit* occurs.

```
203b <ExcludeFiles15 203b>≡ (203a)
    if [ $gitflag -eq 1 ]
    then
        git add debian/copyright
        git commit -am "Changed debian/copyright"
    fi
}
```

<CheckSignature 207a>

31.4.6. Create Debian source file

The script then continues to execute the *BuildNewVersion* function and passes the parameters necessary for exclusion to the *mk-origtargz* program. (Reference to this location in the other function).

In this way, a new orig tarball is created with *mk-origtargz* without the files to be excluded from the previous *.tar.gz and its contents are inserted into the existing Git repository with *gbp import-orig*.

```
203c <BuildNewVersion12 203c>≡ (194b)
    # Creating orig file using mk-origtargz
    if [ -z ${Version1} ]
    then
        whiptail --title "No version number!" \
        --msgbox "No version - no *.orig.tar.gz! Bye!" 15 60
        exit
    fi
    echo "mk-origtargz --package "${SourceName}" \
    --version "${Version1}${AddOpt}" "${UpstreamSourceName}" >> ${log}
    mk-origtargz --package ${SourceName} \
    --version ${Version1}${AddOpt} ${UpstreamSourceName} 2>> ${log}
<BuildNewVersion13 204>
```

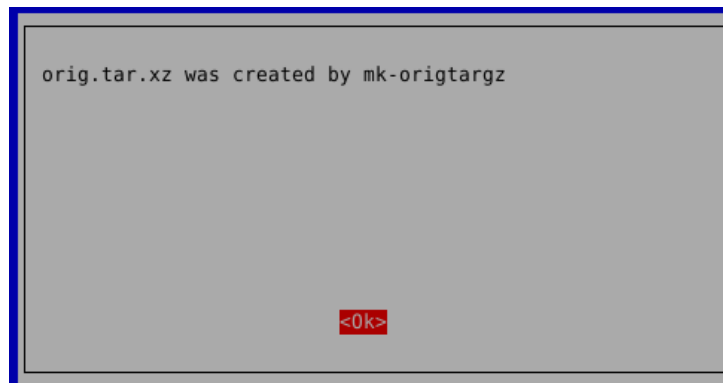


Figure 31.28.: Create orig.tar.xz

```
204  <BuildNewVersion13 204>≡ (203c)
      if [ $? -eq 0 ]
      then
        echo "orig file was created by mk-origtargz" >> ${log}
        Version1=${Version1}${ESuffix}
      else
        echo "mk-origtargz failed" >> ${log}
        whiptail --title "Fatal error" \
          --msgbox "mk-origtargz failed" 15 60
        exit
      fi
```

<BuildNewVersion14 205a>

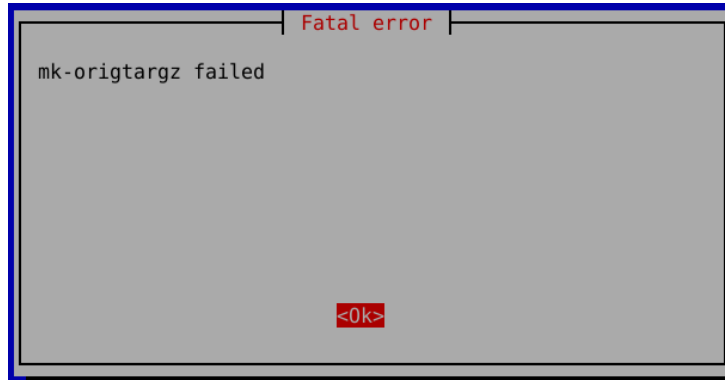


Figure 31.29.: mk-origtargz failed

205a $\langle BuildNewVersion14 \ 205a \rangle \equiv$ (204)

```

Link2File
SearchGbpConf

```

```

cd ${GitPath}

```

```

DebianBranch4Import
 $\langle BuildNewVersion15 \ 218 \rangle$ 

```

If the variable *RecentBranch* does not exist or is empty, it is assigned the value *debian/sid* and a corresponding entry is made in the configuration file.

205b $\langle DebianBranch4Import \ 205b \rangle \equiv$ (284b)

```

function DebianBranch4Import {
    # Called by BuildNewVersion
    # Makes sure RecentBranch contains value
    RecentBranch=$(grep 'RecentBranch=' ${ConfigPath}${OrigName})
    RecentBranch=$(echo ${RecentBranch} | sed --expression='s/RecentBranch=//')
    if [ -z "${RecentBranch}" ]
    then
        RecentBranch="debian/sid"
        changeflag=1
        whiptail --title "Set RecentBranch" \
            --msgbox "Set RecentBranch to ${RecentBranch}" 15 60
    fi

    if [ $changeflag -eq 1 ]
    then
        echo 'RecentBranch=${RecentBranch}' >> ${ConfigPath}${OrigName}
    fi
    echo "Notice from DebianBranch4Import: \
The branch is "${RecentBranch}" >> ${log}
    changeflag=0
}

```

$\langle SpecialExcludeFile \ 199a \rangle$

31.4.7. Verify signature

Some projects publish a signature file along with the source code package. The script can download this and perform a cryptographic check. A signature file in *asc* format is expected.

31.4.7.1. Download signature file

```

206  <DownloadAscFile 206>≡ (207a)
    function DownloadAscFile {
        # Called by BuildNewVersion and itself

        cd ${PrjPath}

        AscFileURL=$(whiptail --title "URL of .asc file" \
            --inputbox "URL and name (with suffix)\nof the .asc file:" \
            --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)

        if [ $? -eq 1 ]
        then
            return 1
        fi

        if [ -z "${AscFileURL}" ]
        then
            echo -e "URL and name (with suffix)\nof the .asc file:"
            read AscFileURL
        fi

        if [ -n "${AscFileURL}" ]
        then
            # getting -asc file using wget
            wget --version ${AscFileURL} &>> ${log} &{
            if [ $? -eq 0 ]
            then
                echo -e "The .asc file was pulled from\n${AscFileURL}\n \
                    by wget." >> ${log}
                whiptail --title "Download successful" \
                    --msgbox "${AscFileURL} was downloaded." 15 60
                if [ -r ${UpstreamSourceName}.asc ]
                then
                    CheckSignature
                else
                    whiptail --title "There is something wrong!" \
                        --msgbox "Maybe you has downloaded the wrong .asc file." 15 60
                    DownloadAscFile
                fi
            else
                DownloadAscFile
            fi
        fi
    fi

```

}

*⟨Link2File 207b⟩***31.4.7.2. Signature Verification**207a *⟨CheckSignature 207a⟩*≡ (203b)

```
function CheckSignature {
    # Called by DownloadAscFile
    gpg --verify ${UpstreamSourceName}.asc >> ${log}

    if [ $? -ne 0 ]
    then
        tail --lines=5 ${log}
        read x
    else
        whiptail --title "Check successfull!" --msgbox "gpg \
        --verify has been successfull" 15 60
    fi
}
```

*⟨DownloadAscFile 206⟩***31.4.8. Replace link with a copy**

The default behavior of *mk-origtargz* (chapter 31.4.6, page 203) is to create a symbolic reference to the original file if it is taken unchanged.

The program script replaces this reference with a corresponding file if necessary.

207b *⟨Link2File 207b⟩*≡ (206)

```
function Link2File {
    # Called by BuildNewVersion
    OrigLinkNr=$(ls -la ${PrjPath} | grep --count -e '.orig.tar.[gx]z -> ')
    if [ $OrigLinkNr -ge 1 ]
    then
        OrigLink=$(ls -la ${PrjPath} | grep --regex='.orig.tar.[gx]z -> ')
        OrigLink=$(echo ${OrigLink} | \ sed --expression='s/^.*:.. //' | sed --expression='s/ //g')
        echo "${OrigLink}" will be transformed into a file" >> ${log}

        LinkTarget=$(echo $OrigLink | sed --expression='s/^.*->/'')
        LinkName=$(echo $OrigLink | sed --expression='s/->.*$/'')
        rm ${PrjPath}/${LinkName}
        cp -a ${PrjPath}/${LinkTarget} ${PrjPath}/${LinkName}
        whiptail --title "Result of transformation link to file:" \
        --msgbox "$(ls -la ${PrjPath})" 15 60
        echo "Result of transformation link to file: \
        "$(ls -la ${PrjPath})" >> ${log}
    fi
}
```

⟨CheckGitStatus 164⟩

31.4.9. *gbp* Configuration File

gbp import-orig (chapter 31.4.10, page 218) adds the downloaded source code to the Git repository.

To control this process, the insertion of a prepared *gbp.conf* file into the *.git* directory is enabled (chapter 31.4.10, page 218)

```
208a  <SearchGbpConf 208a>≡ (215)
      function SearchGbpConf {
          # Called by BuildNewVersion BuildWithUscan

          # Neither .git/gbp.conf nor debian/gbp.conf exist
          if [ ! -f ${GitPath}/.git/gbp.conf -a ! -f ${GitPath}/debian/gbp.conf ]
          then
              if whiptail --title "gbp.conf needed?" \
                  --yesno "Do you want to integrate a special gbp.conf for this project?" \
                  --defaultno --yes-button "Yes" --no-button "No" 15 60
              <SearchGbpConf1 208b>
```

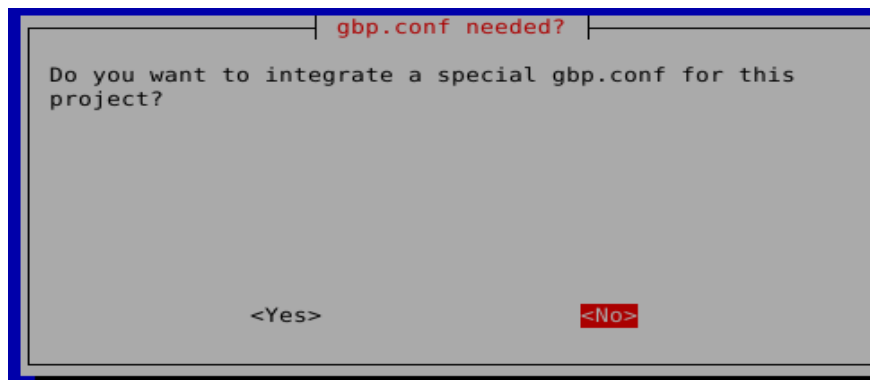


Figure 31.30.: Special gbp.conf

```
208b  <SearchGbpConf1 208b>≡ (208a)
      then
          GbpConfIntegration
      fi
  fi
  <SearchGbpConf2 210>

208c  <GbpConfIntegration 208c>≡ (192b)
      function GbpConfIntegration {
          # Called by SearchGbpConf and itself
          GbpConfPath=$(whiptail --title "gbp.conf" \
              --inputbox "Please insert the path to the your special\n \
              gbp.conf for this project:" \
              --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
          <GbpConfIntegration1 209>
```

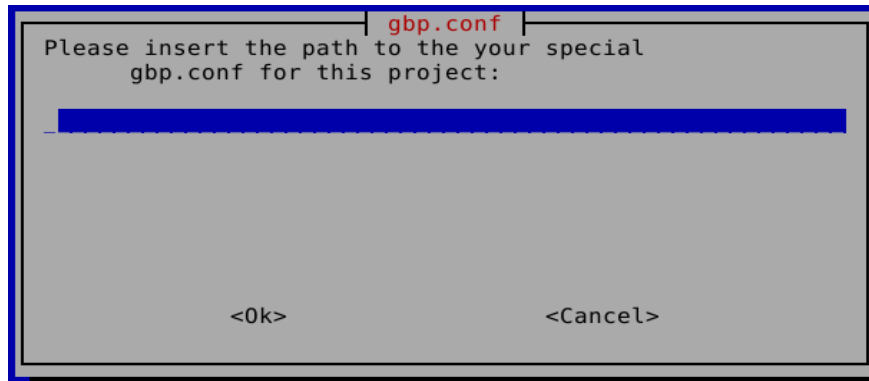


Figure 31.31.: path to the gbp.conf?

```

209  <GbpConfIntegration1 209>≡
      if [ -z "${GbpConfPath}" ]
      then
        echo "Please insert the path to the your special gbp.conf for this project:"
        read GbpConfPath
      fi

      # Replace tilde if necessary
      SuspectPath=${GbpConfPath}
      ReplaceTilde
      GbpConfPath=${CleanPath}

      if [ -f ${GbpConfPath}/gbp.conf ]
      then
        cp -av ${GbpConfPath}/gbp.conf ${GitPath}/.git/
      else
        if whiptail --title "File not found!" \
          --yesno "There was no gbp.conf found at ${GbpConfPath}! Retry?" \
          --yes-button "Yes" --no-button "No" 15 60
        then
          GbpConfIntegration
        fi
      fi
    }

```

(208c)

<TwoConfFilesFound 213a>

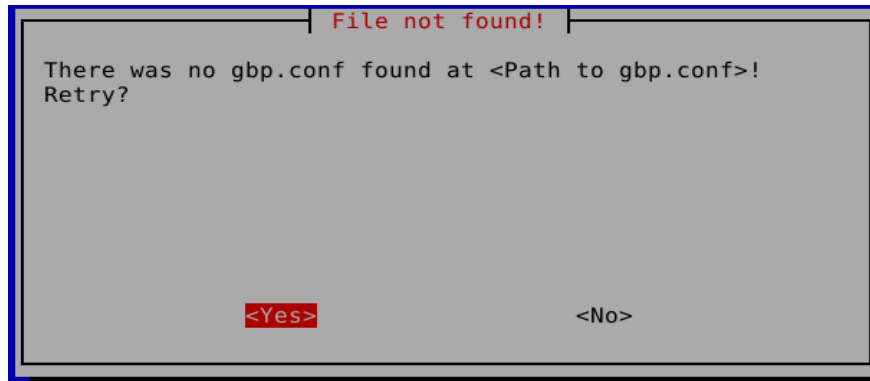


Figure 31.32.: gbp.conf not found

If a *gbp.conf* file is found, it is displayed in the editor for checking and, if necessary, adjustment. Especially the value of the variable *compression* should be checked – especially if a corresponding change of the archive format is to be made.

210 $\langle SearchGbpConf2 \ 210 \rangle \equiv$ (208b)

```
# debian/gbp.conf exists, but not .git/gbp.conf
if [ ! -f ${GitPath}/.git/gbp.conf -a -f ${GitPath}/debian/gbp.conf ]
then
    whiptail --title "Found gbp.conf" \
    --msgbox "Please check and edit your gbp.conf (if necessary)" 15 60
```

$\langle SearchGbpConf3 \ 211 \rangle$



Figure 31.33.: Check gbp.conf

211 $\langle \text{SearchGbpConf3 } 211 \rangle \equiv$ (210)

```

    nano --linenumbers --mouse --softwrap ${GitPath}/debian/gbp.conf
fi
# .git/gbp.conf exists, but not debian/gbp.conf
if [ -f ${GitPath}/.git/gbp.conf -a ! -f ${GitPath}/debian/gbp.conf ]
then
    whiptail --title "Found gbp.conf" \
    --msgbox "Please check and edit your gbp.conf (if necessary)" 15 60

```

$\langle \text{SearchGbpConf4 } 212a \rangle$



Figure 31.34.: Check gbp.conf

212a $\langle \text{SearchGbpConf} 212a \rangle \equiv$ (211)

```

        nano --linenumbers --mouse --softwrap ${GitPath}/.git/gbp.conf
    fi
    # There is a gbp.conf in both directories
    if [ -f ${GitPath}/.git/gbp.conf -a -f ${GitPath}/debian/gbp.conf ]
    then
        TwoConfFilesFound
    fi
}

```

$\langle \text{MovingGbpConfFile} 284b \rangle$

The following is the file with the information that can be used for many Debian packages.

212b $\langle \text{debian/gbp.conf} 212b \rangle \equiv$

```

    # Configuration file for git-buildpackage and friends

    [DEFAULT]
    # use pristine-tar:
    pristine-tar = True
    # generate xz compressed orig file
    compression = xz
    debian-branch = debian/experimental
    upstream-branch = upstream

    [pq]
    patch-numbers = False

    [dch] id-length = 7
    debian-branch = debian/experimental

    [import-orig]
    # filter out unwanted files/dirs from upstream
    filter = [ '.cvsignore', '.gitignore', '.hgtags', '.hgignore', '*.orig', *.rej' ]
    # filter the files out of the tarball passed to pristine-tar
    filter-pristine-tar = True

```


The *TwoConfFilesFound* function handles the case where a file *gbp.conf* occurs in both the *.git/* and *debian/* directories.

213a $\langle TwoConfFilesFound\ 213a \rangle \equiv$ (209)

```
function TwoConfFilesFound {
    # Called by SearchGbpConf MovingGbpConfFile

    whiptail --title "Information" \
        --msgbox "There are a gbp.conf in debian/ and a gbp.conf in .git/" 15 60
 $\langle TwoConfFilesFound0-1\ 213b \rangle$ 
```

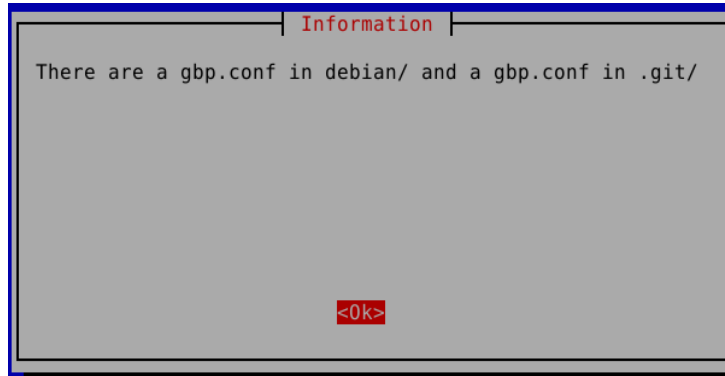


Figure 31.35.: *gbp.conf* found twice.

213b $\langle TwoConfFilesFound0-1\ 213b \rangle \equiv$ (213a)

```
# Are they different?
GitConfFile=$(cat ${GitPath}/.git/gbp.conf)
DebianConfFile=$(cat ${GitPath}/debian/gbp.conf)
if [ "${GitConfFile}" != "${DebianConfFile}" ]
then
    whiptail --title "Warning!" \
        --msgbox "There are a gbp.conf in debian/ and a gbp.conf in .git/\n \
        But they are different!\n\nThe left column is ${GitPath}/.git/gbp.conf\n \
        The right column is ${GitPath}/debian/gbp.conf\n \
        After studying the diff press RETURN!" 15 60
```

$\langle TwoConfFilesFound1\ 214 \rangle$

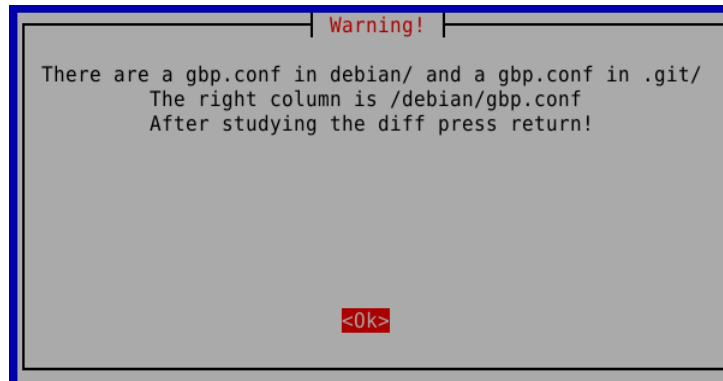


Figure 31.36.: Different configuration files

With `diff --side-by-side (-y)` the difference is displayed in two columns.

```
214  <TwoConfFilesFound1 214>≡ (213b)
      diff --side-by-side ${GitPath}/.git/gbp.conf ${GitPath}/debian/gbp.conf
      read a
      fi
      # Editing
      if whiptail --title "debian/gbp.conf" \
        --yesno "Do you want to edit ${GitPath}/debian/gbp.conf?" \
        --yes-button "Yes" --no-button "No" 15 60
      <TwoConfFilesFound2 215>
```



Figure 31.37.: Do you want to edit *gbp.conf* in the *debian/* directory?

```

215  <TwoConfFilesFound2 215>≡ (214)
      then
        nano --linenumbers --mouse --softwrap ${GitPath}/debian/gbp.conf
      fi
      if
        whiptail --title ".git/gbp.conf" \
          --yesno "Do you want to edit ${GitPath}/.git/gbp.conf (too)?" \
          --yes-button "Yes" --no-button "No" 15 60
      then
        nano --linenumbers --mouse --softwrap ${GitPath}/.git/gbp.conf
      fi
    }

<SearchGbpConf 208a>

```

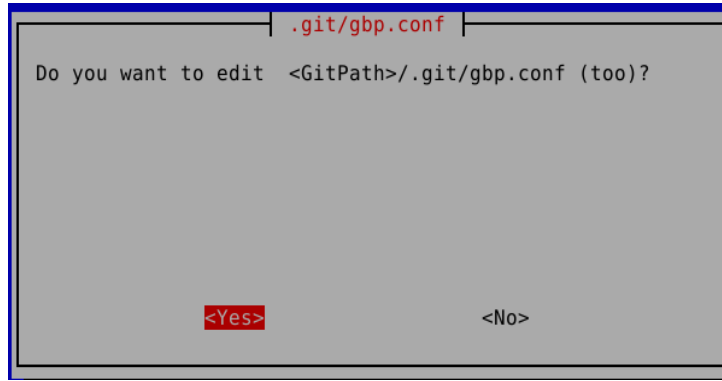


Figure 31.38.: Do you want to edit *gbp.conf* in the *.git/* directory?

A check is made to see if there is still a local change that has not been marked for commit. These must be processed before importing a new version.

Here you can also synchronize with the **Debian** repository on *salsa.debian.org*.

The following section lists the *tags*. For the following commit with *gbp import-orig*, the tag of the version to be imported must not exist yet. (Tag collision)

```
216  <CheckTags 216>≡ (164)
      function CheckTags {
          # Called by BuildWithUscan Import2Git and itself
          # checks git tags before executing gbp import-orig
          echo $(git tag) >> ${log}
          cTags=$(git tag | grep --fixed-strings ${Version1})
          if [ ${#cTags} -gt 0 ]
          then
              cTags1=$(echo ${cTags} | sed --expression='s/ /\n/g')
              if ! whiptail --title "List of dubious tags:" \
                  --yesno "${cTags1}\n\nDo you want to continue regardless?" --defaultno \
                  --yes-button "Yes" --no-button "No" 15 60
              <CheckTags2 217a>
```

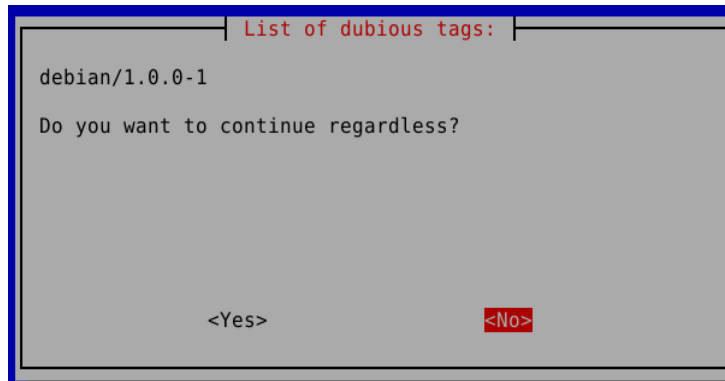


Figure 31.39.: Dubious git tag

If a **Git** tag is listed here, which should still be removed before importing a new version, the answer here is "No". Then the next dialog will ask to remove this **Git** tag.

217a `<CheckTags2 217a>≡` (216)
`then`
`whiptail --title "Delete tags!" \`
`--msgbox "Please delete tags in another terminal\n \`
`and then press ok" 15 60`
`<CheckTags3 217b>`



Figure 31.40.: Delete Git Tags

217b `<CheckTags3 217b>≡` (217a)
`# git tag -d`
`CheckTags`
`fi`
`fi`
`}`
`<Import2Git 219>`

31.4.10. Import to Git

Here *gbp* is used for the first time. As a first step, *gbp import-orig* adds the downloaded source code to the Git repository. The *-debian-branch* option is assigned the contents of the *RecentBranch* variable.

If several possible branches exist, the **Debian** branch to import into can be selected beforehand (chapter 30.4, page 163).

```
218 <BuildNewVersion15 218>≡ (205a)
      Import2Git # Contains import to the git repo using gbp import
      Task=3 # Go to BuildNewRevision
    }

    <DebianFormatTemplate 231b>
```

```

219  <Import2Git 219>≡
function Import2Git {
    # Called by BuildNewVersion and itself

    CheckGitStatus # to exercise caution
    CheckTags

    # Check branch for import
    bl=$(git branch --list | sed --expression='s/* /x_/')
    ba=$(bl)
    for element in ${ba[*]}
    do
        if echo ${element} | grep --quiet '^x_'
        then
            ActiveBranch=$(echo ${element} | sed --expression 's/\x_//')
        fi
    done

    if [ ${ActiveBranch} != ${RecentBranch} ]
    then
        whiptail --title "Check Branch!" \
        --msgbox "The active branch is ${ActiveBranch}.\n\
        In ${ConfigPath}${OrigName} 'RecentBranch' is ${RecentBranch}." \
        15 60
        echo -e "The active branch is "${ActiveBranch}".$n"\
        "In "${ConfigPath}${OrigName}" 'RecentBranch' is "${RecentBranch}".$n\
        FailureNotice
    fi

    # Import to the git repo using gbp import

    echo "Notice from BuildNewVersion: The branch is \
    ${RecentBranch}" >> ${log}
    whiptail --title "Notice from BuildNewVersion:" \
    --msgbox "The branch is ${RecentBranch}" 15 60

    OrigFile=$(ls ${PrjPath}/${SourceName}_${Version1}.orig.tar.?z)
    whiptail --title "Notice from BuildNewVersion:" \
    --msgbox "The orig file to be imported is ${OrigFile}" 15 60
<Import2Git3 220a>

```

Now the input of the passphrase for the GPG key is requested here. If it is not entered promptly, the import is rolled back. It is therefore first queried whether the **GnuPG** key is available (chapter 29.7, page 156). If the question is answered in the negative, the program is terminated.

This signs the Git tag to be generated. This is in accordance with good practice. Only signed tags should be uploaded to *salsa.debian.org*.

Tag signing is the default behavior of *gbp import-orig*. This option was nevertheless included in the command line, because “explicit is better than implicit”.

If you exceptionally do not want to sign, use the option *-no-sign-tags*..

220a *<Import2Git3 220a>*≡ (219)

```
GpgKeyAvailable
echo -e "\n Starting gbp import-orig - Please wait"'\n"
# Signing tags is default
gbp import-orig --verbose --debian-branch=${RecentBranch} \
--sign-tags ${OrigFile}
```

<Import2Git4 220b>

Now *gbp buildpackage* shows in detail all the steps that are now executed. Thereby the determined package version must be confirmed and adjusted.

220b *<Import2Git4 220b>*≡ (220a)

```
if [ $? -eq 0 ]
then
    echo "${OrigFile}" was imported by gbp import-orig" >> ${log}
else
    whiptail --title "Something went wrong!" \
--msgbox "gbp import-orig -v --debian-branch=${RecentBranch} \
--sign-tags ${OrigFile} failed!" 15 60
    echo "gbp import-orig -v --debian-branch=${RecentBranch}" \
--sign-tags "${OrigFile}" failed!" >> ${log}
    FailureNotice
    Import2Git
fi
}
```

<BuildNewVersion 185>

At the end, the *BuildNewVersion* function calls the *BuildNewRevision* function (chapter 32, page 225).

31.5. Download and import with *uscan*

The following steps are performed by the command *gbp import-orig --uscan*

Using the first entry in the file *debian/changelog* (chapter 34.1, page 277), *uscan* determines the version name of the last package built. *uscan* then loads the web page from the *URL* specified in the *debian/watch* file (chapter 32.4.7, page 238). Then *uscan* searches for hyperlinks (*href*) pointing to upstream archives using the search pattern specified in *debian/watch*.

uscan downloads the upstream archive with the latest version if it is newer than the last version specified in *debian/changelog*. The downloaded archive is stored in the parent directory. Finally, *mk-origtargz* (see chapter 18.1.1, page 57) is called.

The program script first checks whether a download with *uscan* is possible and useful.

Furthermore, an existing *gbp.conf* file (with the *SearchGbpConf* function (chapter 31.4.9, page 208)) is opened in the editor for the purpose of checking. Special attention must be paid to the specified compression (*compression*).

```

221 <BuildWithUscan 221>≡ (223a)
    function BuildWithUscan {
        # Called by ClassicalOrUscan

        cd ${GitPath}

        echo "Try gbp import-orig --uscan" >> ${log}
        uscaninfo=$(uscan --no-download --verbose)
        if [ ${#uscaninfo} -gt 0 ]
        then
            SearchGbpConf
            whiptail --title "uscan" --msgbox "${uscaninfo}" \
                --scrolltext 15 60
            echo -e "Result of uscan:\n"${uscaninfo} >> ${log}
            echo ${uscaninfo} | grep '=> Package is up to date' > /dev/null
            if [ $? -eq 0 ]
            then
                whiptail --title "uscan" \
                    --msgbox "Package seems to be up to date.\n \
                    Nothing to do!" 15 60
                echo "Package seems to be up to date. Nothing to do!" \
                    >> ${log}
            <BuildWithUscan4 222a>

```

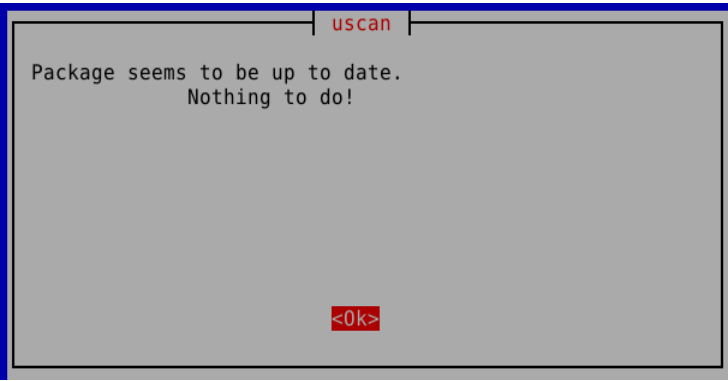


Figure 31.41.: Up to date

222a

```
<BuildWithUscan4 222a>≡
    fi
    echo ${uscaninfo} | grep '=> Newer package available from' \
    > /dev/null
    if [ $? -eq 0 ]
    then
        if ! whiptail --title "Newer package available" \
        --yesno "All well? Continue?" --yes-button "Yes" \
        --no-button "Exit" 15 60
        then
            exit
        fi
    fi
<BuildWithUscan5 222b>
```

(221)

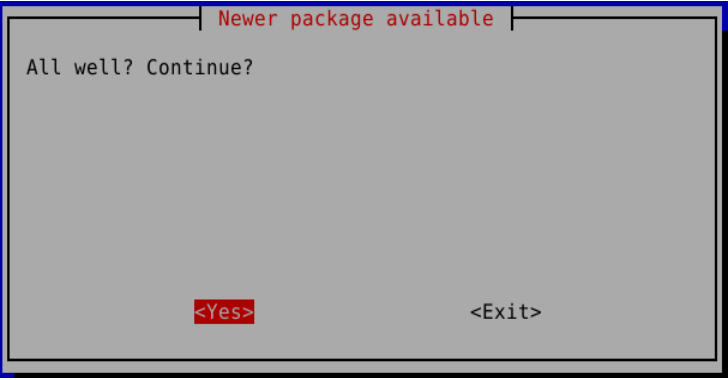


Figure 31.42.: New Version available

222b

```
<BuildWithUscan5 222b>≡
    CheckRepackSuffix
<BuildWithUscan5-1 223b>
```

(222a)

This checks if the *RecentRepackSuffix* entry is present in the configuration file, but no corresponding *debian/watch*.

In this case a hint is given and the file *debian/watch* is presented for editing.

```
223a  <CheckRepackSuffix 223a>≡ (312)
      function CheckRepackSuffix {
          # Called by BuildWithUscan
          if [ -n "${RecentRepackSuffix}" ]
          then
              if ! [ cat ${GitPath}/debian/watch | grep "repacksuffix=" > /dev/null ]
              then
                  whiptail -- title "debian/watch!" \
                      --msgbox "No repacksuffix in debian/watch." 15 60
                  nano --linenumbers --mouse --softwrap ${GitPath}/debian/watch
              fi
          fi
      }

      <BuildWithUscan 221>
```

```
223b  <BuildWithUscan5-1 223b>≡ (222b)
      Version1=$(uscan --no-download --verbose | \
          grep newversion | sed --expression 's/ $newversion = //' )
      CheckGitStatus
      <BuildWithUscan6 223c>
```

To sign the archive of the downloaded source code, the GPG key of the maintainer is required. This must therefore be available. Otherwise it cannot proceed. (Chapter 29.7, page 156)

```
223c  <BuildWithUscan6 223c>≡ (223b)
      GpgKeyAvailable
      <BuildWithUscan7 224>
```

The steps described at the beginning are executed by *gbp import-orig --uscan..*

This is not the way implied in the new-maintainer guild in Chapter 5.21, but uses *gbp import-orig* [37].

```
224  <BuildWithUscan7 224>≡ (223c)
    CheckTags
    # Downloads with uscan and imports with gbp import-orig
    gbp import-orig --uscan --verbose \
    --debian-branch=${RecentBranch} --sign-tags ${OrigFile}
    if [ $? -ne 0 ]
    then
        echo "Import with gbp import-orig --uscan failed!" \
        >> ${log}
        exit
    fi
    echo "Imported with gbp import-orig --uscan" >> ${log}
else
    exit
fi
else
    whiptail -- title "Uscan failed!" \
    --msgbox "Please check the watch file with uscan." 15 60
    echo "Uscan failed! Please check the watch file with uscan." >> ${log}
    exit
fi

Task=3 # Go to BuildNewRevision
}
```

<PrepareUploading (never defined)>

At the end, the *BuildWithUscan* function calls the *BuildNewRevision* function (chapter 32, page 225)

32. Building a new revision

The *BuildNewRevision* function is automatically called by the *BuildNewVersion* and *BuildWithUsca*n functions.

In order to be able to postpone the building of a new revision to a later time, the possibility is given to cancel the program beforehand.

```
225 <BuildNewRevision2 225>≡ (226)
    # Intro
    if ! whiptail --title "New Debian revision" \
    --yesno "A new Debian revision will be built." --yes-button "Yes" \
    --no-button "Exit" 15 60
    then
        exit
    fi

    echo "A new Debian revision will be built." >> ${log}
<BuildNewRevision4 227>
```

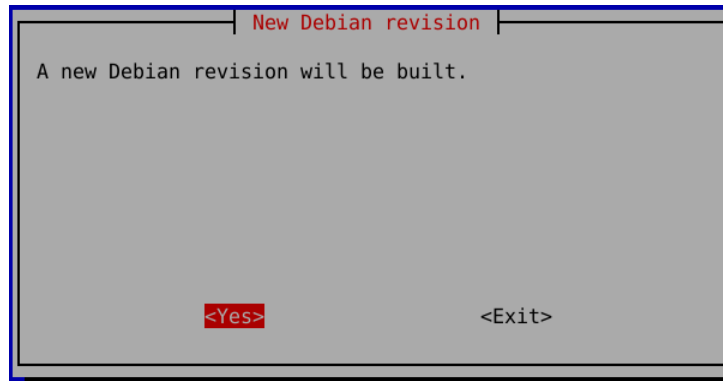


Figure 32.1.: Vuilt new revision

If the *debian/source* directory exists and it is not a **Maven** package, the next question is whether to display the files in the *debian/* directory for editing (chapter 32.3, page 228)

32.1. Creating the Debian directory

If the *debian/source* directory does not (yet) exist, it will be created by the program script. This is usually only relevant if a new package for **Debian** is to be packaged.

226 $\langle BuildNewRevision\ 226 \rangle \equiv$ (294a)

```
function BuildNewRevision {
    # Called by TaskSelect
    cd ${GitPath}

    ## Generate directory if necessary
    echo $(pwd) >> ${log}
    if [ -d debian/source ]
    then
        echo "The directory debian/source in ${GitPath} \
        already exists." >> ${log}
        dfe=1
    else
        mkdir --parents debian/source
        echo "Directory debian/source was created" >> ${log}
        dfe=0
    fi
}
```

$\langle BuildNewRevision2\ 225 \rangle$

The result of the execution of the program script is noted in the log file.

32.2. Request: Build with *mh-make*?

Provided that the **Maven** plugin (chapter 44, page 363) is installed and **Maven** is selected as the build system, you will be asked whether certain files should be created for this build system. Usually this question can be answered in the negative

227 \langle BuildNewRevision4 227 $\rangle \equiv$ (225)

```
# For building java packages with maven

# To avoid an error, if 'MavenPluginFlag' is empty
if [ ! -z ${MavenPluginFlag} ] &&[ ${MavenPluginFlag} -eq 1 ]
then
    if whiptail --title "Maven" \
        --yesno "Should mh_make create the ${PackName}.poms\n \
        file and some maven.* files?\n\n \
        Normally you only need it at the first run" --yes-button "Yes" \
        --no-button "No" --defaultno 15 60
    then
        . ${MavenPluginPath}
        MakeMaven
    fi
fi
```

\langle BuildNewRevision5 228 \rangle

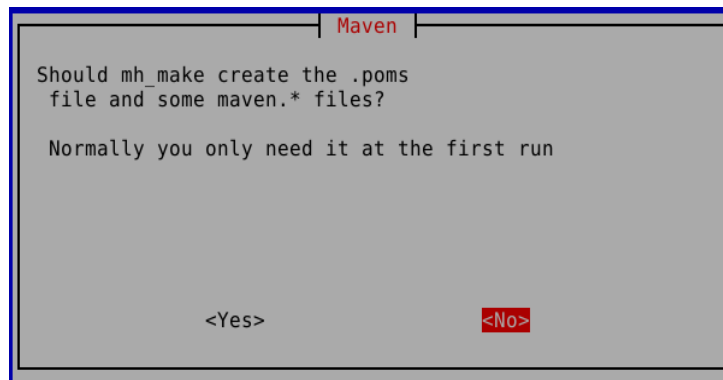


Figure 32.2.: Create data for Maven?

If it is answered in the affirmative, the MavenPlugin is loaded (chapter 44, page 363) and the function *MakeMaven* is called. In chapter 44.3 (page 364) the further steps are described.

32.3. Display the Debian files?

Now the possibility is opened here to create and edit the files in the directory *debian/*.

```

228  <BuildNewRevision5 228>≡ (227)
      # Displaying files in debian/ for editing
      if [ ${dfe} -ne 1 ]
      then
          # DisplayDebianFiles
          DisplayDebianFiles
      else
          if whiptail --title "Showing debian files for editing" \
            --yesno "Should the files of debian/ be displayed\n \
            to check, edit or create them?" --yes-button "Yes" \
            --no-button "No" 15 60
          then
              DisplayDebianFiles
          fi
      fi
      dfe=0

```

<BuildNewRevision5-1 251a>

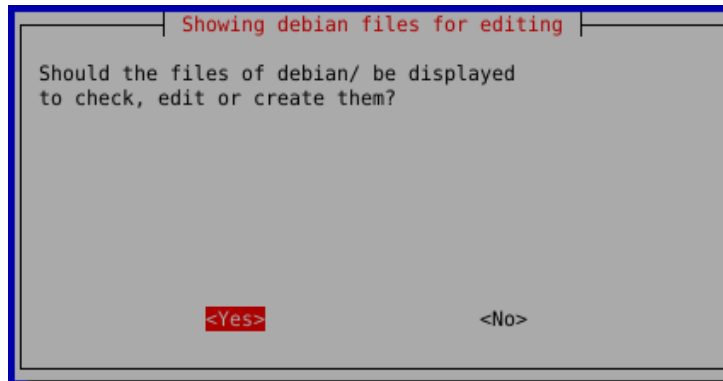


Figure 32.3.: Display Debian files

If the question whether the files in the directory *debian/* should be displayed is answered in the affirmative, these files are created if necessary and displayed for possible editing.

Otherwise, it continues with changes to the upstream code (chapter 33, page 251) and only the *debian/changelog* file is still displayed (chapter 34.1, page 277).

32.4. Files in the directory *debian/*

The files in the *debian/* directory are used to control and document the build process.

The script has the task to create the necessary or more frequently used files in the *debian/* directory - as far as possible - to make suggestions for their content.

32.4.1. Display the Debian files

If the files in the *debian/* directory are to be displayed in order to check them, edit them or even create them, the following function is executed:

```

229 <DisplayDebianFiles 229>≡ (250)
    function DisplayDebianFiles {
        # Called by BuildNewRevision

        # Add Debian files

        # If the Debian files already exists, you can review and improve them now
        # If not, you have to write them

        ## There is default content for Debian files
        ## Change the default values, if you know, what you are doing

        # Loading Webext plugin or Python3 Plugin if needed

        if [ -z "${WebextFlag}" ]
        then
            WebextFlag=0
        fi
    
```

```
if [ ${WebextFlag} -eq 1 ]
then
    . ${WebextPluginPath}
fi
```

```
if [ -z "${PythonFlag}" ]
then
    PythonFlag=0
fi
```

```
if [ ${PythonFlag} -eq 1 ]
then
    . ${PythonPluginPath}
fi
```

DebianFormatTemplate $\langle DisplayDebianFiles1\ 230a \rangle$

s. Chapter 32.4.2, page 231

230a $\langle DisplayDebianFiles1\ 230a \rangle \equiv$ (229)
DebianUpstreamMetadataTemplate
 $\langle DisplayDebianFiles2\ 230b \rangle$

Chapter 32.4.4, Page 232

230b $\langle DisplayDebianFiles2\ 230b \rangle \equiv$ (230a)
DebianCopyrightTemplate
 $\langle DisplayDebianFiles3\ 230c \rangle$

230c $\langle DisplayDebianFiles3\ 230c \rangle \equiv$ (230b)
DebianControlTemplate $\langle DisplayDebianFiles4\ 230d \rangle$

230d $\langle DisplayDebianFiles4\ 230d \rangle \equiv$ (230c)
DebianWatchTemplate
 $\langle DisplayDebianFiles4-1\ 230e \rangle$

230e $\langle DisplayDebianFiles4-1\ 230e \rangle \equiv$ (230d)
DebianRulesTemplate
 $\langle DisplayDebianFiles4-2\ 230f \rangle$

230f $\langle DisplayDebianFiles4-2\ 230f \rangle \equiv$ (230e)
DebianSalsaCiTemplate

$\langle DisplayDebianFiles5\ 247a \rangle$

If a Java package is to be built without a build system, it may be necessary to create a *debian/javabuild* file.

If the Maven plugin is used, the Maven files are displayed for editing (chapter 44.4, page 373).

```
231a  <DisplayDebianFiles10 231a>≡ (248c)

    if [ ${JavaFlag} -eq 1 ]
    then
        if [ ${MavenPluginFlag} -eq 0 ]
        then
            DebianJavabuildTemplate
        else
            ls debian/ | grep 'maven'
            if [ $? -eq 0 ]
            then
                ShowMaven
            fi
        fi
    fi
    CmeFix
}

<ForceOrig 300a>
```

32.4.2. debian/source/format

This file contains the format of the source package. In the *debian/source/format* file, the *3.0 (quilt)* entry is created. This means that it is *not* a native package. In the other case, the *3.0 (native)* entry must be created.

There is a detailed description of this in the *Debian Guide for New Package Maintainers* ¹.

```
231b  <DebianFormatTemplate 231b>≡ (218)

    function DebianFormatTemplate {
        # Called by DisplayDebianFiles

        # String for debian/source/format
        str4format="3.0 (quilt)"

        if ! [ -f ${GitPath}/debian/source/format ]
        then
            touch ${GitPath}/debian/source/format
            echo ${str4format} >> ${GitPath}/debian/source/format
            echo "/debian/source/format was created." >> ${log}
        fi
        nano --linenumbers --mouse --softwrap ${GitPath}/debian/source/format
    }

    <DebianUpstreamMetadataTemplate 232>
```

¹<https://www.debian.org/doc/manuals/maint-guide/dother.en.html#sourcef>

32.4.3. debian/source/include.binaries

Basically, binary files should not find inclusion in the `Debian` package. They should therefore be regularly excluded from inclusion in the `*.orig` archive (chapter 31.4.5, page 195).

However, there are exceptions. These may include media files and compressed documentation. These files must be listed in the `debian/source/include-binaries` file with their path for documentation purposes.

32.4.4. debian/upstream/metadata

There is a detailed description^[38] of this `YAML` file in English. There is also reference to DEP-12².

There it is also explained which information should be entered in the individual lines by the maintainer, if this information is available.

It is recommended to remove the comments when editing the file.

```
232 <DebianUpstreamMetadataTemplate 232>≡ (231b)
function DebianUpstreamMetadataTemplate {
    # Called by DisplayDebianFiles

    # Strings for debian/upstream/metadata
    if ! [ -f ${GitPath}/debian/upstream/metadata ]
    then
        mkdir --parents debian/upstream
        # creating a template for debian/upstream/metadata
        echo -e "# You can find a description at\n#\n\
https://wiki.debian.org/UpstreamMetadata" \
        >> debian/upstream/metadata
        echo "# Archive: " >> debian/upstream/metadata
        echo "# ASCL-id: " >> debian/upstream/metadata
        echo "Bug-Database: " >> debian/upstream/metadata
        echo "Bug-Submit: " >> debian/upstream/metadata
        echo "# Cite-As: " >> debian/upstream/metadata
        echo "Changelog: " >> debian/upstream/metadata
        echo "# CPE: " >> debian/upstream/metadata
        echo "Documentation: " >> debian/upstream/metadata
        echo "# Donation: " >> debian/upstream/metadata
        echo "# FAQ: " >> debian/upstream/metadata
        echo "# Funding: " >> debian/upstream/metadata
        echo "# Gallery: " >> debian/upstream/metadata
        echo "# Other-References: " >> debian/upstream/metadata
        echo "# Reference: " >> debian/upstream/metadata
        echo "#     Author: " >> debian/upstream/metadata
        echo "#     Booktitle: " >> debian/upstream/metadata
        echo "#     DOI: " >> debian/upstream/metadata
        echo "#     Editor: " >> debian/upstream/metadata
        echo "#     Eprint: " >> debian/upstream/metadata
        echo "#     ISBN: " >> debian/upstream/metadata
        echo "#     ISSN: " >> debian/upstream/metadata
```

²Resource:^[39]

```

echo "#      Journal: " >> debian/upstream/metadata
echo "#      Number: " >> debian/upstream/metadata
echo "#      Pages: " >> debian/upstream/metadata
echo "#      PMID: " >> debian/upstream/metadata
echo "#      Publisher: " >> debian/upstream/metadata
echo "#      Title: " >> debian/upstream/metadata
echo "#      Type: " >> debian/upstream/metadata
echo "#      URL: " >> debian/upstream/metadata
echo "#      Volume: " >> debian/upstream/metadata
echo "#      Year: " >> debian/upstream/metadata
echo "#      Debian-package: " >> debian/upstream/metadata
echo "# Registration: " >> debian/upstream/metadata
echo "# Registry: " >> debian/upstream/metadata
echo "Repository: " >> debian/upstream/metadata
echo "Repository-Browse: " >> debian/upstream/metadata
echo "# Screenshots: " >> debian/upstream/metadata
echo "# Security-Contact: " >> debian/upstream/metadata
echo "# Webservice: " >> debian/upstream/metadata
fi
nano --linenumbers --mouse --softwrap ${GitPath}/debian/upstream/metadata
}

```

⟨DebianCopyrightTemplate 233⟩

The following fields are relevant for many packages. Their presence is partially checked by *lintian*.

Bug-Database URL zur Liste der bekannten Fehler

Bug-Submit Adresse, an die Fehlermeldungen gesandt werden können.

Changelog URL des Upstream Changelogs

Documentation Upstream Dokumentation

Repository URL zum Upstream-Repository

Repository-Browse Durchsuchbares Repository von Upstream

32.4.5. debian/copyright

This file contains information about the copyright and licenses of the original authors' sources.

This file can be created with *debmake -cc* and stored in DEP-5 format [17].

233 ⟨DebianCopyrightTemplate 233⟩≡ (232)

```

function DebianCopyrightTemplate {
    # Called by DisplayDebianFiles

    if ! [ -f ${GitPath}/debian/copyright ]
    then
        # creating debian/copyright using debmake
        debmake -cc > debian/copyright
    }
}

```

⟨DebianCopyrightTemplate2 234a⟩

Excerpt from the man page for *debmake*

`-c, --copyright scan source for copyright+license text and exit.`

- `-c`: simple output style
- `-cc`: normal output style (similar to the `debian/copyright` file)
- `-ccc`: debug output style

After that, the file still needs to be edited. Files with the same author and the same license can be combined. If files are under several licenses, these licenses are connected with *or*.

234a $\langle DebianCopyrightTemplate2 \ 234a \rangle \equiv$ (233)

```
fi
    nano --linenumbers --mouse --softwrap ${GitPath}/debian/copyright
}
```

$\langle TeamMaintainer \ 139 \rangle$

Each *Files* paragraph in the machine-readable copyright file must reference a license, each of which has a separate license paragraph. These paragraphs must appear **after** all *Files* paragraphs.

Standalone license paragraphs can be used to provide the full license text for a particular license only once, rather than repeating it in each *Files* section that references it.[40]

If the license text is available at `/usr/share/common-licenses/`, instead of the complete license text an abbreviated version and the file name together with the path of the file containing the license text (for example `/usr/share/common-licenses/GPL-3`) shall be listed.

32.4.6. *debian/control*

This file contains essential values used by the package management tools .

The package management system processes data that is stored in a common format called control data in the *control* file. This data is used for source packages, binary packages, and the **.changes* files that control the installation of the uploaded files.

Details are described in the *Debian* policy[7].

32.4.6.1. Fundamental structure

The program script creates a "framework" of the *control* file for the source package. The *control* file of the binary package and the *.changes* file are created by the build process from the information in the section for the binary file(s).

234b $\langle DebianControlTemplate \ 234b \rangle \equiv$ (144)

```
function DebianControlTemplate {
    # Called by DisplayDebianFiles

    # Strings for debian/control
    str4versiondebhelpers="(=13)"
 $\langle DebianControlTemplate1 \ 235a \rangle$ 
```

Starting with version *debhelper* ≥ 12 , the *compatibility version* is no longer additionally maintained in the *debian/compat* file. Instead, in the *debian/control* file, the *debhelper* entry is replaced by *debhelper-compat* with version ($= 13$).³ This also applies to all subsequent releases.

The use of version 11 is already discouraged.

235a $\langle DebianControlTemplate1 \text{ 235a} \rangle \equiv$ (234b)
`str4standardsversion="4.6.2"`

$\langle DebianControlTemplate2 \text{ 235b} \rangle$

In the file *debian/control* there must be an entry "default-version: " which specifies the conformance to the version of the Debian policy (see chapter 7.2, page 21). Here the current version (*now: 4.6.1*) is given in each case.

At this point in the program script, a template for the *debian/control* file is created if the file does not already exist.

235b $\langle DebianControlTemplate2 \text{ 235b} \rangle \equiv$ (235a)

```
if ! [ -f ${GitPath}/debian/control ]
then
    # creating a template for debian/control
    echo -e "Source: "${SourceName}" > debian/control
    echo -e "Priority: optional" >> debian/control
```

$\langle DebianControlTemplate3 \text{ 236d} \rangle$

Name and email address of **Maintainer** and, if applicable, **Uploaders** are determined by the *DEBValues* function. (Chapter 29.4.1, page 136)

235c $\langle DebianControlTemplate4 \text{ 235c} \rangle \equiv$ (236d)
`DEBValues`

```
if [ -n "${Maintainer}" ]
then
    echo -e "Maintainer: "${Maintainer}" >> debian/control
else
    echo "Maintainer: " >> debian/control
fi

if [ -n "${Uploaders}" ]
then
    echo -e "Uploaders: "${Uploaders}" >> debian/control
fi
```

$\langle DebianControlTemplate5 \text{ 236a} \rangle$

³https://release.debian.org/bookworm/freeze_policy.html (2023)

```
236a  <DebianControlTemplate5 236a>≡ (235c)
      echo -e "Build-Depends: debhelper-compat" \
      ${str4versiondebhelpers} >> debian/control
      <DebianControlTemplate6 237a>
```

```
236b  <DebianControlTemplate7 236b>≡ (237a)
      echo -e "Standards-Version: "${str4standardsversion} \
      >> debian/control
      echo -e "Rules-Requires-Root: no" >> debian/control
      echo -e "Vcs-Git: https://salsa.debian.org/"${SalsaName} \
      >> debian/control
      BrowserName=$(echo ${SalsaName} | sed --expression='s/.git$//g')
      echo -e "Vcs-Browser: https://salsa.debian.org/"${BrowserName} \
      >> debian/control
      echo -e "Homepage: \n" >> debian/control
      <DebianControlTemplate8 236c>
```

Now follows in the file *debian/control* the information about the binary package.

```
236c  <DebianControlTemplate8 236c>≡ (236b)
      echo -e "Package: "${PackName} >> debian/control
      echo -e "Architecture: all" >> debian/control
      echo -e "Depends: \${misc:Depends}" >> debian/control
      echo -e "Description: " >> debian/control
      echo "A template for debian/control was created." >> ${log}
      <DebianControlTemplate9 237b>
```

32.4.6.2. Adaptations for Java packages

Information from this file is used when creating the *control* file of the binary package.

For packaging Java packages, the following entries can already be made..

```
236d  <DebianControlTemplate3 236d>≡ (235b)
      if [ ${JavaFlag} -eq 1 ]
      then
          echo -e "Section: java" >> debian/control
      else
          echo -e "Section:" >> debian/control
      fi
      <DebianControlTemplate4 235c>
```


For packages that are maintained in a team, the address of the team is specified here. This is usually the email address of the mailing list. In these cases the `Uploaders` field must also be filled with the names of the package maintainers.

Maintainer for the Java team, for example, is *Debian Java maintainers* <pkg-java-maintainers@lists.aliases.debian.org>.

So that the packager does not always have to write his full name and e-mail address, the script first looks for this data in the configuration file (chapter 29.4.1, page 136).

```
237a <DebianControlTemplate6 237a>≡ (236a)
      if [ ${JavaFlag} -eq 1 ]
      then
        echo " , default-jdk" >> debian/control

        if [ ${MavenPluginFlag} -eq 1 ]
        then
          echo " , maven-debian-helper" >> debian/control
        fi
      fi
    <DebianControlTemplate7 236b>
```

32.4.6.3. Web-Extension-Plugin

Call the function to customize the *debian/control* file for **Mozilla** add-ons. This function is located in the **Webext** plugin (chapter 45.2.3, page 381)

```
237b <DebianControlTemplate9 237b>≡ (236c)
      if [ ${WebextFlag} -eq 1 ]
      then
        WebextControl
      fi
    <DebianControlTemplate10 237c>
```

32.4.6.4. Python-Plugin

Call the function to customize the *debian/control* file for **Python** packages and libraries. This function is located in the **Python** plugin (chapter 46.2, page 384).

```
237c <DebianControlTemplate10 237c>≡ (237b)
      if [ ${PythonFlag} -eq 1 ]
      then
        PythonControl
      fi
    <DebianControlTemplate11 237d>

237d <DebianControlTemplate11 237d>≡ (237c)
      fi
      nano --linenumbers --mouse --softwrap ${GitPath}/debian/control
    }

    <OptionsWatchFile 239a>
```

32.4.7. debian/watch

The *watch* file in the **Debian** directory contains data for the *uscan* program (chapter 31.5, page 221 and chapter 37.4, page 316)..

To determine the name of the source package, *uscan* reads the first entry in the *debian/changelog* file (chapter 34.1, page 277). Using this entry, *uscan* also determines the version name of the last package built [41].

Then *uscan* processes the lines of the *debian/watch* file in one go from top to bottom. Details about the *debian/watch* file are described in the corresponding article in the Debian wiki⁴.

Regular expressions in Perl format⁵ can be used in this file. Lines starting with a *#* are ignored as comment lines.

At the beginning of this file is the version of the format used. This specification is required. The recommended version number is *4* and is already entered by the script when the file is created.

```
238 <DebianWatchTemplate 238>≡ (241b)
function DebianWatchTemplate {
    # Called by DisplayDebianFiles

    # String for debian/watch str4watch="version=4"

    if ! [ -f ${GitPath}/debian/watch ]
    then
        # creating a template for debian/watch
        echo ${str4watch} > debian/watch
        OptionsWatchFile
    <DebianWatchTemplate3 241c>
```

⁴<https://wiki.debian.org/debian/watch>

⁵see for this:

- https://de.wikibooks.org/wiki/Perl-Programmierung:_Reguläre_Ausdrücke
- <http://www.mathe2.uni-bayreuth.de/perl/GK/regExp.htm>
- http://perl-seiten.privat.t-online.de/html/perl_reg.html

Now the options are defined how *uscan* can check if the current version is built too.

The evaluation of the versioning follows the presentation in chapter 11, (page 35).

The options specify rules for selecting possible Upstream.archives. They are explained in the manual page (man page) of *uscan* ⁶.

The program script sets the options based on the information available so far in the *debian/watch* file.

The program script avoids spaces in the list of options. Otherwise the options list must be framed by double quotes (").

```
239a  <OptionsWatchFile 239a>≡ (237d)
      function OptionsWatchFile {
          # Called by DebianWatchfile

          olf='\\n'
          WOpt='opts='
```

```
<OptionsWatchFile1 239b>
```

When creating the **.orig.tar.** file (chapter 31.4.1, page 185), the original archive format was already determined. This information is now used for the *debian/watch* file..

The following option specifies when building a new version using *uscan* that the **.orig.tar.** file is archived in a different compression format than the upstream archive to be downloaded. Specified here as the compression format of the orig archive is *xz*.

Namely, if the upstream archive is in a zip format (including .xpi, .jar, or .oxz), a repackaging must be performed. The *compression=xz* option specifies that an **.orig.tar.xz* is formed. The *repack* option is dispensable in this case; but explicit is better than implicit.

```
239b  <OptionsWatchFile1 239b>≡ (239a)
      # Repacked <UpstreamPackage>.zip
      RepackFlag=0
      ZipSuffix=(.zip .oxz .xpi .jar)
      if [[ ${ZipSuffix} =~ ${RecentUpstreamSuffix} ]]
      then
          WOpt=${WOpt}${olf}'repack,compression=xz,'
          RepackFlag=1
      fi
```

```
<OptionsWatchFile2 240a>
```

⁶<https://people.debian.org/~osamu/uscan.html#WATCH-FILE-OPTIONS>

If a repackaging has to be done because files have to be removed from the source code archive, this should be evident in the name of the **.orig.tar.** file. The *repacksuffix* option is used for this purpose. The files to be excluded result from the corresponding list (*Files-Excluded*) in the file *debian/copyright*.

```
240a  <OptionsWatchFile2 240a>≡ (239b)
      # Excluded files
      if [ -z ${RecentRepackSuffix} ]
      then
        if [ ${RepackFlag} -eq 1 ]
        then
          WOpt=${WOpt}${olf}'repacksuffix='${RecentRepackSuffix}',\\n'
        else
          WOpt=${WOpt}${olf}'repack,compression=xz,\\n\
          repacksuffix='${RecentRepackSuffix}',\\n'
        fi
```

<OptionsWatchFile3 240b>

Next is the *dversionmangle* option. This normalizes the last found upstream version name in the *debian/changelog* file to match the version of the available upstream archive. This is done by removing the Debian specific suffixes like *+dfsg* or *+ds* by way of substitution.

```
240b  <OptionsWatchFile3 240b>≡ (240a)
      WOpt=${WOpt}${olf}'dversionmangle=s/'${RecentRepackSuffix}'//,'
      fi
```

<OptionsWatchFile4 240c>

The *uversionmangle* option normalizes the strings of the upstream version files extracted from the links to those files in the web page source code. In the version name, the non-numeric characters (except the dot) are replaced in a meaningful way for the sake of uniformity. This makes the version designation of the upstream archive versioning scheme compliant (chapter 11.2, page 35). This is used as version sort index when selecting the latest upstream version.

```
240c  <OptionsWatchFile4 240c>≡ (240b)

      # For beta-, rc- etc. releases

      WOpt=${WOpt}${olf}'uversionmangle=s/-?([^\d.]+)/~$1/;tr/A-Z/a-z/,,'
      <OptionsWatchFile5 241a>
```

filenamemangle generates the upstream tarball filename from the selected *href* string if the comparison patterns can extract the latest upstream version from the selected *href* string. Otherwise, the upstream tarball filename is generated from its full URL string and the missing upstream version is inserted from the generated upstream tarball filename.

Without this option, the default upstream tarball filename is generated by taking the last component of the URL and removing everything after a '?' or '#'.

```
241a  <OptionsWatchFile5 241a>≡ (240c)
      # For packages from Github
      if echo ${DownloadUrl} | grep "github" > /dev/null
      then
          WOpt=${WOpt}${olf}'filenamemangle=s/.+\/v?(\d\S+)\.*/$1/, '
      fi
```

<OptionsWatchFile8 241b>

```
241b  <OptionsWatchFile8 241b>≡ (241a)
      # If there are no options
      if [ $#WOpt -eq 6 ]
      then
          WOpt='# '${WOpt} WOpt=$(echo ${WOpt} | sed 's/\\/\\/\'')
      fi

      echo -e ${WOpt} >> debian/watch
  }
```

<DebianWatchTemplate 238>

uscan loads the web page from the *URL* specified in *debian/watch*. Then *uscan* searches for hyperlinks (*hrefs*) pointing to upstream archives using the search pattern specified in *debian/watch*.

Starting from the URL with which the source code was downloaded, the program script defines how to search for a new version. The URL was assigned to the variable *DownloadURL* as value (chapter 31.4.1, page 185).

```
241c  <DebianWatchTemplate3 241c>≡ (238)
      if echo ${DownloadUrl} | grep "github" > /dev/null
      then
          DownloadUrl=$(echo ${DownloadUrl} | \
          sed --expression 's/archive.*$/releases/')
          DownloadUrl=${DownloadUrl}' ./v?(\d\S+)\.tar.gz'
          echo -e ${DownloadUrl} >> debian/watch
      fi
      echo "A template for debian/watch was created." >> ${log}
      whiptail --title "Edit debian/watch!" \
      --msgbox "Please insert reasonable regular expressions\n \
      into debian/watch!" 15 60
      fi
      nano --linenumbers --mouse --softwrap ${GitPath}/debian/watch
  }
```

<DebianRulesTemplate 242>

April 6, 2025

An example:

```
opts=repack,compression=xz,dversionmangle=s/\+dfsg$//,\
uversionmangle=s/-Beta/~beta/;s/-rc/~rc/,\
filenamemangle=s/.*\/v?(\d+\.\d+\.\d+(?:-(Beta|rc)\d+)?)\.\tar\
.gz/jax-maven-plugin-$1.tar.gz/ \
https://github.com/davidmoten/jax-maven-plugin/releases .*/v?(\d\S+)\.\tar\.\gz
```

The *debian/watch* file, when in the directory where the Git repository is located, can be checked with the *uscan --no-download --debug* command. The *--no-download* option causes a found, newer upstream archive not to be downloaded. The *--debug* option generates a human readable report which also shows the status of the internal variables.

32.4.8. *debian/rules* - Fundamental structure

This file controls the flow of the build process.

Unlike the other files in the *debian* directory, the *debian/rules* file must be marked as executable.

Like any other *makefile*, the *debian/rules* file is composed of several rules that define the target and how these rules are executed. In the Debian policy, chapter 4.9[7] *Main building script: debian/rules* the details are explained.

32.4.8.1. Create the file

If the file *debian/rules* does not exist yet, it will be created.

It is a Makefile and therefore has a corresponding *Shebang* (*#!/usr/bin/make -f*).

```
242 <DebianRulesTemplate 242>≡ (241c)
function DebianRulesTemplate {
    # Called by DisplayDebianFiles

    # Strings for debian/rules
    str4rules="#!/usr/bin/make -f\n# -*- makefile -*-\n"\
    str4rulesdh=""

    %:\n\tdh \${@}\n\n"
    if ! [ -f ${GitPath}/debian/rules ]
    then
        touch ${GitPath}/debian/rules
        echo -e ${str4rules} >> ${GitPath}/debian/rules
    <DebianRulesTemplate1 243a>
```

32.4.8.2. Export of variables

It is turned on the more comprehensive output of the messages by exporting the variables *DH_VERBOSE* and *DH_OPTIONS*, as well as assigning appropriate values to them.

```
243a <DebianRulesTemplate1 243a>≡ (242)
      echo -e "# Uncomment this to turn on verbose mode.\n" \
      >> ${GitPath}/debian/rules
      echo -e "export DH_VERBOSE=1\nexport DH_OPTIONS=-v\n" \
      >> ${GitPath}/debian/rules
```

<DebianRulesTemplate2 243b>

In addition, supplementary variables are exported for various package types. These are defined in the respective plugins. The description for **Java** packages can be found in chapter 43.1 (page 361). The customization for Java packages is done by the *Rules4Java* function.

```
243b <DebianRulesTemplate2 243b>≡ (243a)
      if [ JavaFlag -eq 1 ]
      then
          Rules4Java
      fi
```

<DebianRulesTemplate3 243c>

Building the **Mozilla** extensions also requires additional entries. The description of the specifics for the **Mozilla** add-ons can be found in chapter 45.2.2 (page 379).

The customization for *Webext* packages is done by the *WebextRules* function in the corresponding plugin (chapter 45 page 377).

```
243c <DebianRulesTemplate3 243c>≡ (243b)
      if [ ${WebextFlag} -eq 1 ]
      then
          WebextRules
      fi
```

<DebianRulesTemplate4 243d>

The same applies to the **Python** packages (chapter 46.1 page 384).

```
243d <DebianRulesTemplate4 243d>≡ (243c)
      if [ ${PythonFlag} -eq 1 ]
      then
          PythonRules
      fi
```

<DebianRulesTemplate5 243e>

32.4.8.3. Call of the Debhelper

```
243e <DebianRulesTemplate5 243e>≡ (243d)
      echo -e ${str4rulesdh} >> ${GitPath}/debian/rules
```

<DebianRulesTemplate6 244a>

For **Maven** packages, the call to the *debhelper* must be added. This is done by the *Rules3MavenDH* function of the Maven plugin (chapter 44.5 page 376).

```
244a  <DebianRulesTemplate6 244a>≡ (243e)
      if [ ${MavenPluginFlag} -eq 1 ]
      then
          Rules4MavenDH
      fi

      <DebianRulesTemplate7 244b>
```

Also for the **Mozilla** extensions the call to the *debhelper* must be added. This is done by the *Web extension plugin* (chapter 45, page 377).

```
244b  <DebianRulesTemplate7 244b>≡ (244a)
      if [ ${WebextFlag} -eq 1 ]
      then
          WebextRulesDH
      fi

      <DebianRulesTemplate10 244c>
```

Also, when packaging Python packages, add the *debhelper* call to the *debian/rules* file.

```
244c  <DebianRulesTemplate10 244c>≡ (244b)
      if [ ${PythonFlag} -eq 1 ]
      then
          PythonRulesDH
      fi

      <DebianRulesTemplate11 245>
```


32.4.8.4. *debian/rules* - overrides

Sometimes it is necessary to execute additional steps before or after the execution of the respective *debhelper* scripts. For this purpose, the respective script is overridden.

Example:

```
override_dh_auto_build:
    dh_auto_build -- -f org/xmlunit/pom.xml package -DskipTests
```

The first line of the example names the target to be modified. The second line must start with a *tab character* in a width of 8 characters.

The double minus sign means that first the parameters are executed that *dh_auto_build* normally passes. After that, other parameters may be listed that are passed to the program.

package denotes the so-called *goal*.

32.4.8.5. End of the function

At the end of the function, an entry is made in the log file. The *debian/rules* file is displayed for editing. Finally it is made executable as *Make* file.

```
245 <DebianRulesTemplate11 245>≡ (244c)
    echo "debian/rules was created " >${log}

    fi
    nano --linenumbers --mouse --softwrap ${GitPath}/debian/rules

    if ! [ -x ${GitPath}/debian/rules ]
    then
        chmod ugo+x ${GitPath}/debian/rules
        echo "${GitPath}/debian/rules is now executable" >> ${log}
    fi
}

<DebianSalsaCiTemplate 246a>
```

32.4.9. salsa-ci.yml

On *salsa.debian.org*, enter *debian/salsa-ci.yml* in the respective project under *Settings - CI/CD - General pipelines - CI/CD configuration file*, or the corresponding file name located in the directory *debian/*.

This triggers the automatic build process for the *Reproducible Builds* as well.

```
246a  <DebianSalsaCiTemplate 246a>≡ (245)
      function DebianSalsaCiTemplate {
          # Called by DisplayDebianFiles

          # String for debian/salsa-ci.yml
          str4salsa="include:\n\
- https://salsa.debian.org/salsa-ci-team/pipeline/raw/master/salsa-ci.yml\n\
- https://salsa.debian.org/salsa-ci-team/pipeline/raw/master/pipeline-jobs.yml"

          if ! [ -f ${GitPath}/debian/salsa-ci.yml ]
          then
              touch ${GitPath}/debian/salsa-ci.yml
              echo -e ${str4salsa} >> ${GitPath}/debian/salsa-ci.yml
              echo "debian/salsa-ci.yml was created." >> ${log}
          fi
          nano --linenumbers --mouse --softwrap ${GitPath}/debian/salsa-ci.yml
      }

      <SelectChangesFile 313>
```

32.4.10. debian/javabuild

The following function allows the creation of a *debian/javabuild* file, which can be used to build a Java package without a build system.

The *debian/javabuild* file contains on each line the name of a *.jar* file followed by a list of source code files or directories. This file is read by the *java-helper* program *jh_build*.

```
246b  <DebianJavabuildTemplate 246b>≡ (281a)
      function DebianJavabuildTemplate {
          # Called by DisplayDebianFiles
          # For building a java package without build system (like maven)
          if [ -f debian/javabuild ]
          then
              if whiptail --title "Creating debian/javabuild?" \
              --defaultno --yesno "Should debian/javabuild be created?" \
              --yes-button "Yes" --no-button "No" 15 60
              then
                  echo "# NameOfJarFile SourceDirToPackage" >>debian/javabuild
                  echo "debian/javabuild was created" >> ${log}
                  nano --linenumbers --mouse --softwrap debian/javabuild
              fi
          fi
      }

      <DisplayDebianChangelog 277b>
```

32.4.11. <Package name>.install

This file is needed to specify where a file should be installed. It is mandatory to make sure that the package name is really the name of the binary to be built.

```
247a <DisplayDebianFiles5 247a>≡ (230f)
      if [ ${WebextFlag} -eq 1 ] &&[ ! -f ${GitPath}/debian/${PackName}.install ]
      then
          WebextInstall
      fi

      nano --linenumbers --mouse \
          --softwrap ${GitPath}/debian/${PackName}.install
<DisplayDebianFiles6 247b>
```

Example:

```
nc.jar usr/share/java
```

This copies the respective files to the future directory. By default, it is not possible to rename a file so that the file name matches the naming of Java libraries in the *Debian Policy for Java* [26], for example.

As described in the manual for *dh_install* ⁷. This then also requires further steps.

- The package must have a build dependency on *dh-exec*. The package that must be specified in the *debian/control* file is *dh-exec*
- The installation file must be marked as executable.

32.4.12. <Package name>.dirs

This file does not necessarily have to have the name of the binary. The package name can also be omitted completely

```
247b <DisplayDebianFiles6 247b>≡ (247a)
      nano --linenumbers --mouse \
          --softwrap ${GitPath}/debian/${PackName}.dirs
<DisplayDebianFiles7 247c>
```

Example:

```
usr/share/java
```

32.4.13. <Package name|.docs

The files listed here are installed in the build process from the corresponding *debhelper* *dh_installdocs* to a directory created for this purpose */usr/share/docs/<package name>*.

The *LICENSE* file does not need to be included. This is automatically installed after */usr/share/docs/<package name>*.

```
247c <DisplayDebianFiles7 247c>≡ (247b)
      nano --linenumbers --mouse \
          --softwrap ${GitPath}/debian/${PackName}.docs
<DisplayDebianFiles8 248a>
```

⁷https://manpages.debian.org/unstable/debhelper/dh_install.1.en.html

32.4.14. <Package name>.links

This file is called from the *dh_link*.

```
248a  <DisplayDebianFiles8 248a>≡ (247c)
      if [ ${WebextFlag} -eq 1 ] &&[ ! -f ${GitPath}/debian/${PackName}.links ]
      then
          WebextLinksTB
      fi

      nano --linenumbers --mouse \
          --softwrap ${GitPath}/debian/${PackName}.links
      <DisplayDebianFiles9 248c>
```

32.4.15. <Package name>.desktop

```
248b  <PacketnameDesktop 248b>≡
      [Desktop Entry]
      X-AppInstall-Package=JVerein
      X-AppInstall-Popcon=1
      X-AppInstall-Section=main

      Version=1.0
      Name=JVerein
      Comment=Administration of an Association
      Comment[de]=Vereinssoftware

      Exec=jameica
      Icon=jameica-icon
      Terminal=false
      Type=Application
      Categories=Office
      Keywords=Association;Verein
      StartupNotify=true
```

32.4.16. <Package name>.examples

This file is called by the *dh_installexamples*.

```
248c  <DisplayDebianFiles9 248c>≡ (248a)
      nano --linenumbers --mouse \
          --softwrap ${GitPath}/debian/${PackName}.examples

      <DisplayDebianFiles10 231a>
```

32.4.17. README.Debian

32.4.18. README.source

This file can be used to document the exclusions (chapter 10.4.1.3, page 32) and their justifications.⁸ This file gets an entry in any case when building with *maven*. (Chapter 44.4.5, page 375.

32.5. Checking the files in *debian/* with CmeFix

The *cme fix dpkg* command checks the *dpkg* files, updates obsolete parameters, and applies any fixes.

With the parameter *-verbose* you get more information about what happens. The parameter *-backup* creates backup files before saving the changes. These are identified by the extension *.old*. Note that in this case the long form of the options are really just prefixed with a hyphen⁹.

249 $\langle CmeFix\ 249 \rangle \equiv$ (283)

```
function CmeFix {
    # Called by DisplayDebianFiles

    if whiptail --title "Check and fix with cme?" \
        --yesno "Should debian files be checked and fixed using 'cme fix'?" \
        --yes-button "Yes" --no-button "No" 15 60
    then
        if whiptail --title "Backup?" \
            --yesno "Should the recent files be backuped (recommended)?" \
            --yes-button "Yes" --no-button "No" 15 60
        then
            cme fix -verbose -backup dpkg
        else
            cme fix -verbose dpkg
        fi
    fi
}
 $\langle cme\ fix\ 250 \rangle$ 
```

⁸Debian-Policy, chapter 4.14 [7]

⁹<https://manpages.debian.org/unstable/cme/cme.1p.en.html>

Execution of the program script is halted here so that the output of the *cme fix dpkg* command can be analyzed. In another terminal – if the backup option was selected – the original files can be compared with the newly created files, backup files can be deleted and corrections can be made if necessary.

```
250  <cme fix 250>≡ (249)
      echo "Please check the result of cme fix!"
      echo "You can check and fix it in another terminal."
      echo "Please press RETURN to go on."
      read a
    fi
  }

  <DisplayDebianFiles 229>
```

33. Making changes to upstream code

We continue with possible changes to the source code. If no changes are made to the source code, the treatment of the file *debian/changelog* (chapter 34.1, page 277) follows directly.

```
251a  <BuildNewRevision5-1 251a>≡ (228)
      # Patches treatment
      PatchesTreatment
```

```
<BuildNewRevision6 277a>
```

When a new revision is created, the user is asked if changes should be made to the source code. However, to do this, the program first checks again (see chapter 31.2, page 177) whether a directory *debian/patches* already exists and informs the user of the result of this check.

```
251b  <PatchesTreatment 251b>≡ (303a)
      function PatchesTreatment {
          # Called by BuildNewRevision

          # Patches treatment
          cd ${GitPath}
          if [ -d debian/patches ]
          then
              whiptail --title "Info" \
                  --msgbox "There is a directory debian/patches" 15 60
      <PatchesTreatment1 252a>
```

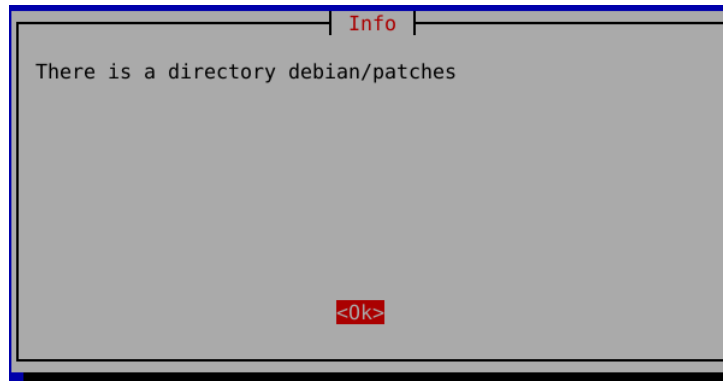


Figure 33.1.: There is a directory debian/patches.

```
252a  <PatchesTreatment1 252a>≡ (251b)
      else
        whiptail --title "Info" \
          --msgbox "There is no directory debian/patches" 15 60
      fi

<PatchesTreatment2 252b>
```

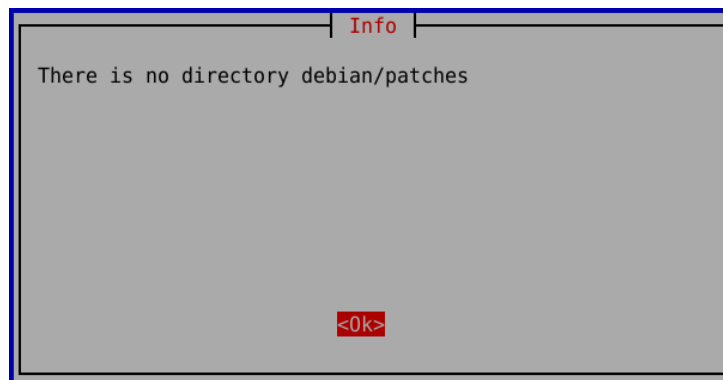


Figure 33.2.: There is no directory debian/patches.

Now the query occurs whether a patch is to be created, edited or deleted. Furthermore, you can choose between the two methods *quilt* (chapter 33.2, page 263) and *gbp pq* (chapter 33.1, page 254). Of course, the build process can also be continued without *patch* (chapter 34.1, page 277).

```
252b  <PatchesTreatment2 252b>≡ (252a)
      PMTask=$(whiptail --title "Tasks:" \
        --radiolist "Do you want to create, edit or delete patches?" 15 60 4 \
        "0" "By using quilt" off \
        "1" "By using gbp pq" off \
        "2" "No" on --cancel-button "Exit" 3>&2 2>&1 1>&3)

<PatchesTreatment3 253a>
```

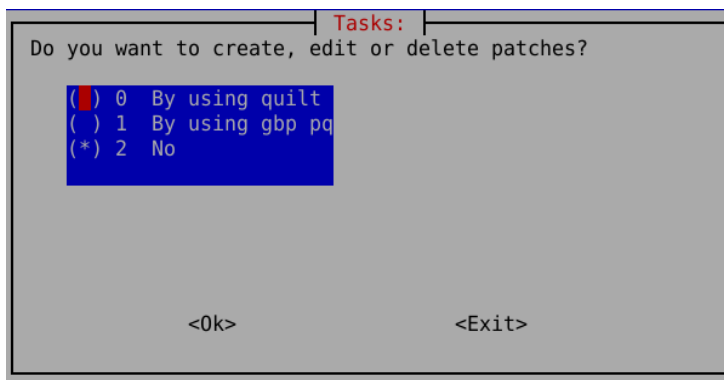



Figure 33.3.: Create patches for Debian

If the question is answered in the negative, the next step is to edit the *debian/changelog* file (chapter 34.1, page 277).

With *Exit* the program script is left.

```
253a  <PatchesTreatment3 253a>≡ (252b)
      if [ -z "${PMTask}" ]
      then
          exit
      fi
```

<PatchesTreatment4 253b>

If you have decided to patch and to use a method, the program script now calls the respective further functions.

```
253b  <PatchesTreatment4 253b>≡ (253a)
      if [ ${PMTask} -eq 0 ]
      then
          PatchRunNr=0
          PatchTasks
      elif [ $PMTask -eq 1 ]
      then
          CheckGitStatus
          PQMigration
      fi
  }
```

<LastQuestionsBeforeBuild 290b>

Either you work with *gbp pq* or with *quilt* (chapter 33.2, page 263).

If one has decided to work with *quilt*, the variable *PatchRunNr* is set to 0 and the function *PatchTasks* (chapter 33.2, page 263) is called.

Otherwise, the *PQMigration* function (chapter 33.1.1, page 254) of the program script is called.

33.1. Working with *gbp pq*

To manage patches, *gbp pq* can be used. This is mainly intended for source packages in 3.0 (*quilt*) format (see chapter 32.4.2, page 231). The modification of the upstream source code is done by files in the *debian/patches/* directory.

The *PQMigration* function can create a patch queue branch using these patches if it does not already exist. Furthermore, this function can update an existing patch queue branch.

33.1.1. Creating a Patch Queue Branch

It will check again if there is a file *debian/patches/series* or not.

It also checks if there is a matching *patch queue branch*.

There are four cases to distinguish:

1. There is neither a file *debian/patches/series* nor a matching *patch-queue branch*. Then *git checkout -b* creates a new *patch-queue branch* and changes to it.
2. The file *debian/patches/series* does not exist, but a matching *patch-queue branch* already does. Then its existence is pointed out and changed to it.
3. The file *debian/patches/series* exists, but no matching *patch-queue branch*. Then *gbp pq import* is used to create a matching *patch-queue branch*. However, this involves significant risks if not all patches in the queue are applicable (chapter 31.2, page 177). Therefore, it is **urgently** recommended to create the *patch queue branch* if necessary before downloading a new version. If this has not been done, the option to work with *quilt* should be selected.
4. Both the *debian/patches/series* file and the matching *patch-queue branch* exist. This will often be the case. Then *gbp pq rebase* is applied first.

A *patch queue* branch is created only in cases 1 and 3.

```

254  <PQMigration 254>≡ (258)
      function PQMigration {
          # Called by PatchesTreatment and itself
          # Transfers patches into patch-queue branch
          npqf=0
          cd ${GitPath}
      }

      if [ ! -f debian/patches/series ]
      # debian/patches/series does not exists
      then
          # Case 2
          # Anything is easy
          npqf=1

```

```

if echo $(git branch) | grep --quiet 'patch-queue/${RecentBranch}
then
    whiptail --title "PQ-branch exists" \
    --msgbox "Branch 'patch-queue/${RecentBranch}' exists." 15 60
    git checkout patch-queue/${RecentBranch}
(PQMigration0 255a)

```

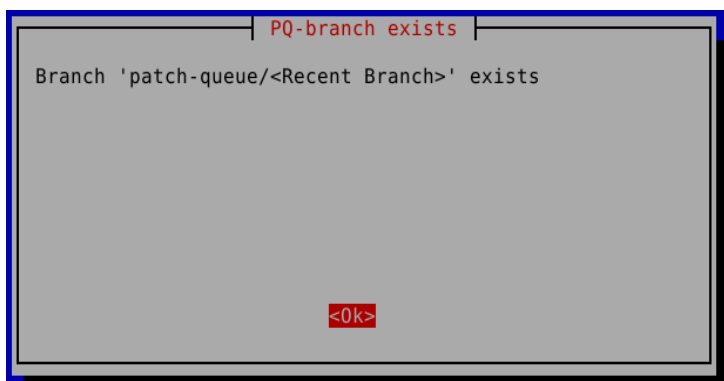


Figure 33.4.: There exists a PQ branch.

If the file *debian/patches/series* does not exist and no matching patch queue branch exists, it is recreated here and changed to this.

```

255a (PQMigration0 255a)≡ (254)
    else
        git checkout -b patch-queue/${RecentBranch}
    fi
(PQMigration1 255b)

```

You can then immediately start editing the source code (chapter 33.1.7, page 260).

Both *gbp pq rebase* and *gbp pq import* assume that there are no unversioned files in the current Git branch and that all previous patches can be applied. Otherwise, merge conflicts need to be resolved.

The user is informed about this.

```

255b (PQMigration1 255b)≡ (255a)
    else
        # debian/patches/series exists
        Notice="All patches listed in 'debian/patches/series' \n\
        have to be appliable"'!'" \n\
        Otherwise you have to solve 'merge conflicts'"!'"
        if whiptail --title "Attention please"'!'" \
        --yesno "${Notice} Do you want to check the situation?" \
        --yes-button "Yes" --no-button "No" 15 60
        then
            ShowPatches "check"
(PQMigration1-1 256)

```

What can be done here is given in chapter 33.5, page 273).

```

256 <PQMigration1-1 256>≡ (255b)
    if whiptail --title "Attention please"!!' \
      --yesno "Do you want to edit 'debian/patches/series'?" \
      --yes-button "Yes" --no-button "No" 15 60
    then
      nano --linenumbers --mouse --softwrap debian/patches/series
    fi

    echo -e ${Notice}
    CheckGitStatus
  fi
<PQMigration2 257a>

```



Figure 33.5.: Hint on the requirements for further work.

33.1.2. Manual Editing

In the *debian/patches/series* file, yes, all patches are listed in the order in which they have been applied so far. Manual edit

Now they need to be checked to see if the problems have already been improved by upstream, making these patches obsolete.

Then these patches are to be removed from the *debian/patches* directory and their entries from the *debian/patches/series* file. These adjustments are made in the main branch.

33.1.3. Troubleshooting hints

Only general information can be given for troubleshooting after a failure.

If changes to source code files already exist, these are to be undone with *git restore <path/filename>*. A possibly existing directory *.pc/* is to be deleted.

Changes to the *debian/changelog* file are to be committed.

33.1.4. Refreshing the patch queue branch

The program script checks whether a corresponding patch queue branch already exists.

If a suitable patch queue branch exists, the program script first calls the *RebasePQBranch* function.

Otherwise, patch queue branch is created by *gbp pq import* from the current *Git* branch (chapter 33.1.6, page 259).

257a `<PQMigration2 257a>≡` (256)

```
if [ ${npqf} -eq 0 ]
then
  if echo $(git branch) | grep --quiet 'patch-queue/'${RecentBranch}
  # patch-queue branch exists
  then
    # Case 4
    RebaseCounter=0
    RebasePQBranch
```

`<PQMigration3 259>`

The *RebasePQBranch* function executes a *gbp pq rebase*. The commits of the *debian/* branch are applied to the patch queue branch.

With *gbp pq rebase* it is changed to the patch queue branch which is connected to the current branch. All new additions to the current branch are transferred to the patch queue branch by a *rebase*.^[42]

257b `<RebasePQBranch 257b>≡` (269d)

```
function RebasePQBranch {
  # Called by PQMigration and itself
  if [ ${RebaseCounter} == 0 ]
  then
    gbp pq rebase --verbose
  else
    git rebase --continue --verbose
  fi
```

`<RebasePQBranch1 257c>`

If the execution of *gbp pq rebase* fails, manual intervention can and must be performed to rectify the situation (chapter 33.1.5, page 258).

For this purpose, *Git* gives the user hints. These look like the following, for example:

Resolve all conflicts manually, mark them as resolved with Hinweis: "git add/rm <conflicted_f

Manually resolve all conflicts, mark them as resolved with "git add/rm <conflicted_files>" and

257c `<RebasePQBranch1 257c>≡` (257b)

```
if [ $? -ne 0 ]
then
  git rebase --show-current-patch | cat
```

`<RebasePQBranch2 258>`

At this point, the patch that cannot be applied (completely) is displayed. With this help, it is then possible to manually resolve the merge conflicts after the interruption.

```
258 <RebasePQBranch2 258>≡ (257c)
    Notice="gbp pq rebase failed"'\n\
    All changes must be committed"'\n\
    All patches have to be applicable"'\n"
    FailureNotice ${Notice}
    RebaseCounter=$(expr ${RebaseCounter} + 1)
    RebasePQBranch
    fi
}

<PQMigration 254>
```

After that, a new attempt can be made.

33.1.5. Hints for cleaning up the patch queue

In very many cases the *pq rebase* command fails, because there are also always changes from upstream to already patched files. Patches that have already been adopted by the upstream project can be removed.

The program script git in such cases the hint to switch to another terminal. There you can check with *git status* that there is an incomplete rebase.

The following command displays the failed patch in the additional terminal.

```
git rebase --show-current-patch
```

In yet another terminal, the (still) existence of the file to be patched can be checked with the following command.

```
tar --list --file ../<UpstreamPackageName>.orig.tar.xz | grep <Filename>
```

If the file does not (longer) exist, the patch can be removed with

```
git rebase --skip
```

Then, in the (original) terminal in which the program script is running, continue with *RETURN*. This first executes a *git rebase -continue -verbose*.

If the file to be patched exists, it must be edited in an editor. This changes the patches so that they can be applied to the new version.

The file is then to be saved and a *git add* executed. In the original terminal, continue with *RETURN*.

If necessary, repeat the procedure until the program script continues with its execution. Eine Möglichkeit ist es auch, durch den Befehl

```
gbp pq import --time-maschine=<n>
```

solange Commit für Commit durchzugehen, bis die Patch-Queue angewandt werden kann.

n gibt dabei die maximale Anzahl der Rückschritte an.

33.1.6. Import of existing patches

The contents of *debian/patches* are imported into the patch queue branch with *gbp pq* and changed into it¹.

On import, output like this is generated:

```
gbp:info: Trying to apply patches \under 'aaa1011bfd5aa74fea43620aae94709de05f80be' \apply.
```

```
gbp:info: 18 patches listed in 'debian/patches/series' \
imported under 'patch-queue/debian/sid'.
```

It imported each patch file with *gbp pq* and changed to the newly created patch queue branch *patch-queue/debian/sid*.

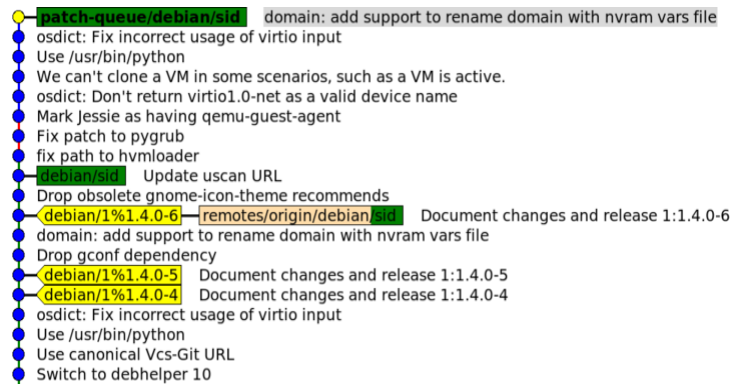


Figure 33.6.: The patch queue branch with patches from *debian/patches* has been applied.[3]

If the file *debian/patches/series* exists, but no corresponding patch queue branch exists yet, *gbp pq import* is executed. This is done in the *PQImport* function (chapter 31.2, page 177).

gbp pq import creates a new patch queue branch from the *quilt patches* in the *debian/patches* directory listed in the *debian/patches/series* file. These patches must be applicable without fuzziness.

```
259 <PQMigration3 259>≡ (257a)
    else
        # patch-queue branch does not exist
        # Case 3
        if whiptail --title "Attention"'' \
        --yesno "Do you really want to use 'gbp pq import'?" \
        --yes-button "Yes" --no-button "Working with Quilt" 15 60
        then
            PQImport 0
        else
            PatchTasks
            # Working with Quilt
        fi
```

¹s. section *gbp.patches.html* and *man.gbp.pq.html*[3]

```
fi
fi
⟨PQMigration4 260⟩
```

The above troubleshooting instructions also apply here (chapter 33.1.3, page 256).

33.1.7. Edit source code

```
260  ⟨PQMigration4 260⟩≡ (259)

# Starting the work in the patch-queue branch echo echo "Break for patching in another terminal" ech

⟨PQMigration5 262a⟩
```


In the separate terminal, `git branch -v` should be used to determine that the correct Git branch (*patch-queue/* branch) is active and that its state matches that of the previous Git branch. Then files in the patch queue branch can be edited.

Code can be added, changed or removed. The patches can be created or “brought into shape” like this.

In doing so, the changes are to be committed as small as possible. What will later become a single patch in *debian/patches/* is simply added by a commit.

The first line of the commit message later becomes part of the patch name. The following lines contain the details about the functions of the patch. Therefore, it is useful to write multiline commit messages.

The multiline commit message should then conform to the *patch tagging guideline* in the patch² The header of the patch file will look like this:

The author and the date are set automatically. The name entered under *Gbp-PQ* must not contain spaces. If necessary, these are to be replaced by *underscores* (`_`). This will be the name of the patch file in the *debian/patches* directory.

```
From: Autor <email address>
Date: <Date and time of creation>
Subject: <Commit message>
```

```
Gbp-Pq: Name <name of the patch>
Forwarded: Yes/No <if necessary>
```

```
* posix/regcomp.c (re_compile_fastmap_iter): Rewrite COMPLEX_BRACKET handling.
```

```
Origin: upstream, http://sourceware.org/git/?p=glibc.git;a=commitdiff;h=bdb56bac
Bug: http://sourceware.org/bugzilla/show_bug.cgi?id=9697
Bug-Debian: http://bugs.debian.org/510219
```

This can be corrected with `git commit --amend -m "New Message"` can be corrected.

33.1.8. Export the patches

Once we are satisfied with the commits, let's regenerate the patches in *debian/patches/* using *gbp pq*. This will switch you back to the *debian/sid* branch and regenerate the patches using a method similar to `git-format-patch`:

The result could now be added as follows:

```
gbp pq export
git add debian/patches
git commit
```

gbp pq export means:

Exported the patches on the patch queue branch associated with the current branch to a quilt row of patches to *debian/patches/* and updated the *series* file in the current branch (e.g., *debian/sid*)³[3].

²[https://dep-team.pages.debian.net/deps/dep3/\[24\]](https://dep-team.pages.debian.net/deps/dep3/[24])

³<https://honk.sigxcpu.org/projects/git-buildpackage/manual-html/gbp.patches.html>

To avoid having to transfer the result by hand each time, *--commit* can also be passed to the *gbp pq export* command.

262a $\langle PQMigration5\ 262a \rangle \equiv$ (260)

```
# Export
if whiptail --title "Use gbp pq export?" \
--yesno "Do you like to use 'gbp pq export --commit'?" \
--yes-button "Yes" --no-button "No" 15 60
 $\langle PQMigration7\ 262b \rangle$ 
```

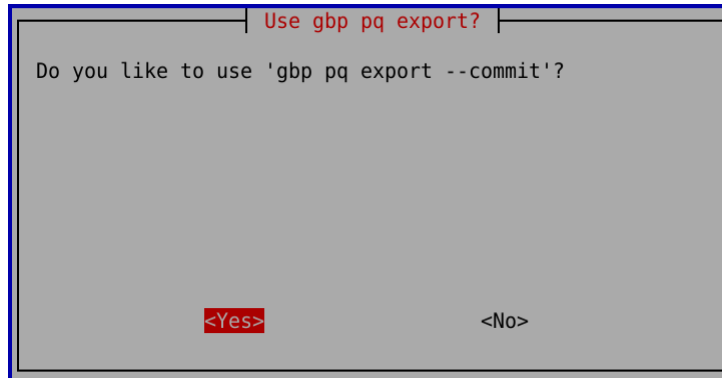


Figure 33.7.: Should *gbp pq export* be applied?

262b $\langle PQMigration7\ 262b \rangle \equiv$ (262a)

```
then
    gbp pq export --commit --verbose >> ${log} 2>&1
    echo "Check branch."
    git branch --verbose
    echo "Press RETURN to continue!"
    read a
 $\langle PQMigration8\ 262c \rangle$ 
```

After exporting the patches, the next step is usually to determine the distribution. (Chapter 34.3.1, page 285)

If the question is answered in the negative, manual intervention is possible.

262c $\langle PQMigration8\ 262c \rangle \equiv$ (262b)

```
else
    git log
    git checkout ${HistoricBranch}
    ReplaceConfigLines 'RecentBranch' ${HistoricBranch}
    git branch
    echo "Break for importing in another way in another terminal"
    echo "After finishing press RETURN to go on!"
    read a
fi
}
 $\langle PatchTasks\ 264a \rangle$ 
```

33.2. Using Quilt

The changes to the upstream are made with the help of Quilt. As described at the beginning, *dquilt* has to be set up first (chapter 18.5, page 70).

The setup in *.bashrc* cannot be used when running this program (chapter 18.5, page 70).

The program therefore first checks whether *quilt* is available. Since the script ignores the alias of *.bashrc*, *dquilt* is defined in a variable.

```
263a  <CreateDquilt 263a>≡ (269a)
      function CreatedQuilt {
          # Called by PatchTasks
```

```
      # Check whether quilt is available if [ ${PatchRunNr} -eq 0 ] then if [ ! -f ~/.quiltrc-dpkg ] <Creat
```

The line checks if there is a file *.quiltrc-dpkg* in the user's home directory. If this file does not exist, the message *dquilt* is not configured, because then the corresponding alias for *quilt --quiltrc=\$HOME/.quiltrc-dpkg* cannot be created.

```
263b  <CreateDquilt1 263b>≡ (263a)
      then
          whiptail --title "No dquilt!" \
          --msgbox "Dquilt seems not to be configured.\n \
          See: https://www.debian.org/doc/manuals/maint-guide/modify.html" \
          15 60
          exit
      fi
```

```
<CreateDquilt2 263c>
```

If the file *.quiltrc-dpkg* is not found, the script prints an appropriate message and exits. Only if the file */.quiltrc-dpkg* exists, the following definition can be made.

```
263c  <CreateDquilt2 263c>≡ (263b)
      # Definition of dquilt
          dquilt="quilt --quiltrc=${HOME}/.quiltrc-dpkg"
      fi
  }
```

```
<MakePatches 265b>
```

A selection is made as to whether patches are to be created, edited or removed. These tasks can be executed in any order and also several times.

```

264a  <PatchTasks 264a>≡ (262c)
      function PatchTasks {
          # Called by PatchesTreatment and itself

          cd ${GitPath}
          if [ ! -d debian/patches ]
          then
              # Create patch
              if whiptail --title "Patches" --yesno "Is a patch necessary?" \
                --yes-button "Yes" --no-button "No" 15 60
              then
                  CreateDquilt
                  MakePatches
              else
                  return
              fi
          else
              CreateDquilt
              PTask=$(whiptail --title "Tasks:" \
                --radiolist "What do you like to do? " 15 60 8 \
                "0" "Display patch files to check or edit them" off \
                "1" "Create additional patch" off \
                "2" "Add another patch to existing patch file" off \
                "3" "Show patch files for deleting" off \
                "4" "Edit debian/patches/series" off \
                "5" "Exit to go on" on --cancel-button "Cancel" 3>&2 2>&1 1>&3) <PatchTasks1 264b>

```

If the *Cancel* button is pressed, the *Exit to go on* task is also executed. Here the value 1.

```

264b  <PatchTasks1 264b>≡ (264a)

      if [ $? -eq 1 ]
      then
          PTask=6
      fi

```

<PatchTasks2 264c>

In the following section, commands are assigned to the individual tasks. Thereby the variable *PTask* contains a digit from 0 to 4.

For example, if this variable is assigned the value *0 = Display patch files to check or edit them*, then the *ChangePatches* function is executed.

In the following *case statement*, *0*) is a value from the list of 0 to 4 against which the contents of the variable *PTask* are compared.

```

264c  <PatchTasks2 264c>≡ (264b)
      # Patches treatment
      case "$PTask" in
          0) ChangePatches;; # Edit patches
      <PatchTasks3 265a>

```

Using the *JaxWS* package, we describe how to edit existing patches.

33.2.1. Create patch

```

265a  ⟨PatchTasks3 265a⟩≡ (264c)
        1) MakePatches;; # If (more) patches are necessary
        ⟨PatchTasks4 269b⟩

        Now the creation of a patch begins.

265b  ⟨MakePatches 265b⟩≡ (263c)
        function MakePatches {
            # Called by PatchTasks and itself

            cd ${GitPath}
            cnpr=0 CreateNewPatch
            if [ $cnpr -ne 0 ]
            then
                return
            fi

            PatchRunNr=1

            if whiptail --title "Another patch?" \
            --yesno "Do you want to apply another patch?" --yes-button "Yes" \
            --no-button "No" 15 60
            then
                MakePatches
            fi
        }

        ⟨ShowPatches 273⟩

```

33.3. Create new patch

266a *⟨CreateNewPatch 266a⟩*≡ (267b)

```
function CreateNewPatch {
    # Called by MakePatches

    PatchFileName=$(whiptail --title "Patch name" \
--inputbox "Name of the patchfile:" \
--cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
    if [ $? -eq 1 ]
    then
        return 1
    fi

    # Because a user might use blanks in a filename
    PatchFileName=$(echo ${PatchFileName} | sed --expression='s/ /-/g')
    if [ -z "${PatchFileName}" ]
    then
        cnpr=1
        return 1
    fi
}
```

⟨CreateNewPatch1 266b⟩

Now it checks if a file `<i0>debian/copyright</i0>` exists.

266b *⟨CreateNewPatch1 266b⟩*≡ (266a)

```
if [ -f debian/patches/${PatchFileName} ]
then
    if ! whiptail --title "Patch exists" \
        --yesno "${PatchFileName} exists already.\nContinue?" \
        --yes-button "Yes" --no-button "No" 15 60
    then
        return 1
    fi
fi
```

⟨CreateNewPatch2 267a⟩

The command *dquilt new* with the parameter of the patch name creates the first "patch file".

The following files are created:

```
debian/patches/series contains the name of the patch DescriptionNoRegistration.patch
.pc/applied-patches contains the name of the patch DescriptionNoRegistration.patch
.pc/DescriptionNoRegistration.patch/
.pc/quilt_patches contains debian/patches
.pc/quilt_series contains series
.pc/.version contains 2
```

If patches already exist, make sure that they are named in the correct order in the *debian/patches/series* file.

```
267a  <CreateNewPatch2 267a>≡ (266b)
      # Create a new patch file
      $dquilt new ${PatchFileName}
    }

    # Patch FileToPatch <CreateNewPatch4 269a>
```

33.4. Select file for patching

```
267b  <FileToPatch 267b>≡ (268)
      function FileToPatch {
        # Called by CreateNewPatch PatchTasks and itself

        FileSelector ${GitPath} # Select the file to be patched
        File2Patch=${selected}

        echo "Patch ${PatchFileName} because of ${File2Patch}" >> ${log}

        # quilt add must get only the filename without
        # the path of the file to be patched
        File2pName=$(basename ${File2Patch})
        $dquilt add -P ${PatchFileName} ${File2pName}

        nano --linenumbers --mouse --softwrap ${File2Patch}

        $dquilt refresh ${PatchFileName}

        if whiptail --title "Patch another" \
          --yesno "Do you want to patch another file in ${PatchFileName}?" \
          --yes-button "Yes" --no-button "No" --defaultno 15 60
        then
          FileToPatch
        fi
      }

    <CreateNewPatch 266a>
```

The file to be corrected is selected in a file selection dialogue.

```

268  <FileSelector 268>≡ (272)
function FileSelector {
    # Called by CreateNewPatch and itself
    # Dialog to select a file using whiptail

    StartPath=$1
    cd $StartPath
    txta=$(ls -a)

    i=0
    flist=''
    for element in ${txta[*]}
    do
        if [ $element == '.' ]
        then
            i=$(expr $i + 1)
            continue
        fi
        flist=$flist' '$i' '${element} i=$(expr $i + 1)
    done

    sel=$(whiptail --title "Filepicker" \
    --menu "Select:" 15 60 6 $flist 3>&2 2>&1 1>&3)

    if [ $? -ne 0 ]
    then
        return
    fi

    # Go back
    if [ ${txta[$sel]} = '..' ]
    then
        cd ..
        StartPath=$(pwd)
        selected=${StartPath}
    else
        selected=${StartPath}/${txta[$sel]}
    fi

    # The order of the following if-clauses is important
    if [ -f ${selected} ]
    then
        if ! whiptail --title "Your choice;" \
        --yesno "${selected}\nContinue?" --yes-button "Yes" \
        --no-button "No" 15 60
        then
            FileSelector ${StartPath}
        fi
    fi

```



```

    if [ -d ${selected} ]
    then
        FileSelector ${selected}
    fi
}

```

⟨FileToPatch 267b⟩

If all files are present in this directory, *dquilt add* with the parameter of the file to be edited will include it in *quilt*.

With this in *.pc/DescriptionNoRegistration.patch/* the mentioned file is added. There it is then available for the first adjustment.

269a ⟨CreateNewPatch4 269a⟩≡ (267a) PatchHeader }

⟨CreateDquilt 263a⟩

269b ⟨PatchTasks4 269b⟩≡ (265a)

```

    2) FileToPatch # Add patch to patch file
    PatchRunNr=1
    $dquilt refresh;; ⟨PatchTasks6 269c⟩

```

33.4.1. Delete Patch

The following selection deletes existing patches. This is necessary if a previous patch is no longer needed because the corrections have been incorporated by the upstream in the meantime.

269c ⟨PatchTasks6 269c⟩≡ (269b)

```

    3) DeletePatches;; # Delete patches
    ⟨PatchTasks7 270b⟩

```

However, it may be that the previous patch needs to be adapted for the current upstream version. One way to do this is to delete the old patch and create a new one (see chapter 33.2.1, page 265).

269d ⟨DeletePatches 269d⟩≡ (270a)

```

function DeletePatches {
    # Called by PatchTasks and itself

    cd ${GitPath}
    DeletePatch
    PatchRunNr=1

    if whiptail --title "Another patch?" \
    --yesno "Do you want to delete another patch?" --yes-button "Yes" \
    --no-button "No" 15 60
    then
        DeletePatches
    fi
}

```

⟨RebasePQBranch 257b⟩

April 6, 2025

If the question is answered with *No*, the program jumps back to the patch task selection (chapter 33.2, page 263).

```
270a  <DeletePatch 270a>≡ (275)
      function DeletePatch {
        # Called by DeletePatches

        ShowPatches "delete" # String will be found in ${1}

        if [ -z "${PatchFileName}" ]
        then
          PatchTasks
        fi

        less --LINE-NUMBERS ${GitPath}/debian/patches/${PatchFileName}

        if whiptail --title "Delete this patch?" \
        --yesno "Do you really want to delete ${PatchFileName}?" \
        --yes-button "Yes" --no-button "No" 15 60
        then
          $dquilt delete -r --backup ${PatchFileName}

          if whiptail --title "Delete backup file?" \
          --yesno "Do you want to delete the backup file, too" \
          --yes-button "Yes" --no-button "No" 15 60
          then
            rm ${GitPath}/debian/patches/${PatchFileName}~
          fi
        fi
      }
```

<DeletePatches 269d>

In order to use the *gbp pq* function, it must be set up beforehand (chapter 10.4.2.2, page 33).

```
270b  <PatchTasks7 270b>≡ (269c)
      # Edit series
      4) nano --linenumbers --mouse --softwrap debian/patches/series;;
      <PatchTasks8 271>
```

33.4.2. Restore the initial state

If a patch has been created, edited or deleted with *quilt*, exiting the function will remove the patches from the upstream code and return it to its original state. This is done with *quilt pop -a*. This means that all applied patches are removed.

```

271  <PatchTasks8 271>≡ (270b)
        5) if [ ${PatchRunNr} -eq 1 ]
            then
                # remove all patches and return the source
                # to its original state
                ${dqilt} pop -a
                PatchRunNr=0
            fi
            # If debian/patches/series is empty,
            # delete directory debian/patches
            if ! [ -s debian/patches/series ]
            then
                rm debian/patches/series
                rmdir debian/patches
            fi
            return;;
        esac
    fi

PatchTasks }

<GitBranch2RecentBranch 289b>

```

So it is possible for Quilt to always create the diff between the previous version and the current version. After that the change is made. The registration information is removed. With *dquilt refresh* one creates the diff to document the change. With *dquilt header -e* now the description of the change is created in the \$EDITOR.

```
272 <PatchHeader 272>≡ (313)
function PatchHeader {
    # Called by CreateNewPatch EditPatch

    PatchDescription=$(whiptail --title "Describe patch!" \
        --inputbox "Description:\n\n" \
        --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
    if [ -z "${PatchDescription}" ]
    then
        echo "Please insert the description of the patch!"
        read PatchDescription
    fi
    if whiptail --title "Describe patch!" --yesno "Forwarded:?" \
        --yes-button "Yes" --no-button "No" 15 60
    then
        PatchForwarded="Yes"
    else
        PatchForwarded="No"
    fi

    DEBValues
    if [ -z ${Uploaders} ]
    then
        PatchAuthor=${Maintainer}
    else
        PatchAuthor=${Uploaders}
    fi

    PatchUpdate=$(date +
%Y'-'%m'-'%d)
    touch ${PatchFileName}.header
    echo "Description: "${PatchDescription} >> ${PatchFileName}.header
    echo "Forwarded: "${PatchForwarded} >> ${PatchFileName}.header
    echo "Author: "${PatchAuthor} >> ${PatchFileName}.header
    echo "Last-Update: "${PatchUpdate} >> ${PatchFileName}.header

    $dquilt header -a < ${PatchFileName}.header
    rm ${PatchFileName}.header
}

<FileSelector 268>
```

Here is an example of this:

Description: removed registration of license Forwarded: No Author: Mechtilde Stehmann <ooo@me

33.5. Patch selection

```

273  <ShowPatches 273>≡
function ShowPatches {
    # Called by EditPatch DeletePatch and itself
    actionstr=${1}

    patchfilesa=$(ls debian/patches)
    i=0; slct=''
    for element in ${patchfilesa[*]}
    do
        if [ ${element} != 'series' ]
        then
            slct=$slct' '$i' '${element}' off '
            newPFA[$i]=${element}
            i=$(expr $i + 1)
        fi
    done

    PatchFileNo=$(whiptail --title "Select patch" \
        --radiolist "Select one of these patches:" \
        --cancel-button "Cancel" 15 60 8 \
        $slct 3>&2 2>&1 1>&3)

    if [ "${actionstr}" = "check" ]
    then
        # Code for PQMigration
        if [ -z "${PatchFileNo}" ]
        then
            return
        fi
        PatchFileName=${newPFA[$PatchFileNo]}
        less --LINE-NUMBERS debian/patches/${PatchFileName}
        ShowPatches "check"
    else
        if [ -z "${PatchFileNo}" ]
        then
            PatchFileName=""
        else
            PatchFileName=${newPFA[$PatchFileNo]}
            if ! whiptail --title "${PatchFileNo}" \
                --yesno "Do you want to ${actionstr} ${PatchFileName}?" \
                --yes-button "Yes" --no-button "No" 15 60
            then
                ShowPatches
            fi
        fi
    fi
}

```

(265b)

April 6, 2025

```
fi
}
```

⟨EditPatch 274⟩

33.6. Editing Patch

```
274  ⟨EditPatch 274⟩≡ (273)
      function EditPatch {
          # Called by ChangePatches

          ShowPatches "edit" # String will be found in ${1}

          if [ -z "${PatchFileName}" ]
          then
              return 1
          else
              PatchRunNr=1
          fi

          $dquilt pop ${PatchFileName}
          nano --linenumbers --mouse --softwrap debian/patches/${PatchFileName}
          $dquilt refresh

          if whiptail --title "New Patch Header" \
              --yesno "Do you want to create a new patch header?" --yes-button "Yes" \
              --no-button "No" 15 60
          then
              PatchHeader
          fi

          while $dquilt push
          do
              $dquilt refresh
          done }

⟨ChangePatches 275⟩
```

33.7. Modify Patch

```

275  <ChangePatches 275>≡ (274)
      function ChangePatches {
          # Called by PatchTasks and itself

          cd ${GitPath}
          EditPatch

          if whiptail --title "Another patch?" \
            --yesno "Do you want to edit another patch?" --yes-button "Yes" \
            --no-button "No" 15 60
          then
              ChangePatches
          fi
      }

      <DeletePatch 270a>

```


34. Building

The actual building of the binary packages is done in a *chroot*. This mainly involves checking that all the required build dependencies are listed in the *debian/control* file (chapter 32.4.6, page 234) and are already available as **Debian** packages. This ensures that the package can also be built reproducibly by the FTP masters without access to further network resources.

34.1. *debian/changelog*

Before the actual build, the *debian/changelog* file is displayed for adjustment.

```
277a  <BuildNewRevision6 277a>≡ (251a)
      # Check debian/changelog
      # - includes creating changelog using gbp dch
      DisplayDebianChangelog
```

```
<MovingGbpConf 284a>
```

First, the script checks if a file *debian/changelog* already exists. If this is the case, this file is displayed and queried if it is correct.

Otherwise it is created with *gbp dch*.

```
277b  <DisplayDebianChangelog 277b>≡ (246b)
      function DisplayDebianChangelog {
          # Called by BuildNewRevision

          newChangelog=0
          if [ -f debian/changelog ]
          then
              less --LINE-NUMBERS debian/changelog
              if ! whiptail --title "Changelog ok?" --defaultno \
                  --yesno "Is debian/changelog ok?" --yes-button "Yes" \
                  --no-button "No" 15 60
              then
                  newChangelog=1
              fi
          else
              newChangelog=1
          fi
      }
```

```
<DisplayDebianChangelog1 278>
```

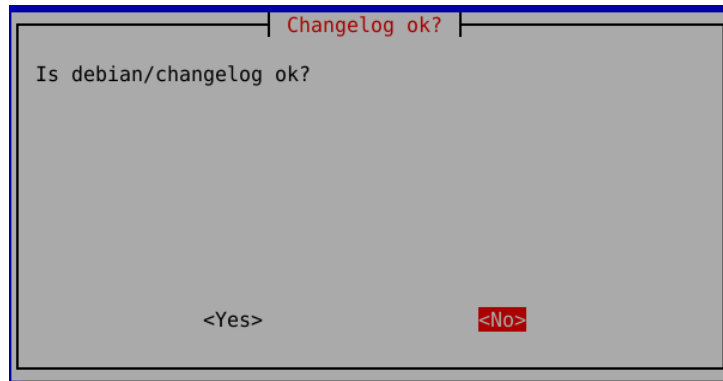


Figure 34.1.: Debian-Changelog OK?

If the question about the correctness of the changelog is answered in the affirmative, the process continues with a possible move of the *gbp* configuration file (chapter 34.2, page 284). Otherwise, this file is displayed in the editor.

Pay special attention to the version and revision designation in the first line of the *debian/changelog* file. This applies mainly when files have been excluded from the upstream source code (chapter 10.4.1.3, page 32).

For a non-maintainer upload (NMU), the first entry expected after the version information is the entry.

* Non-maintainer upload

Also for *dquilt* the file *.bashrc* has to be added. (Chapter <n0>, page <n1>.

```
278 <DisplayDebianChangelog1 278>≡ (277b)
    # Check whether d/control exists without a comment in line 1
    if [ "${newChangelog}" -eq 1 ]
    then
        if ! [ -f debian/control ]
        then
            DebianControlTemplate
        else
            cat --number debian/control | grep '1' | grep '#' > /dev/null
            if [ $? -eq 0 ]
            then
                DebianControlTemplate
            fi
        fi
    fi
    # creating changelog using gbp dch

AddVersionNumber <DisplayDebianChangelog3 282>
```

34.1.1. Insert version number

gbp dch requires the current version name to be specified. If the corresponding variable does not contain this value, you will be prompted to (manually) enter this version designation.

In contrast, *dch* can create the next higher version designation with *dch -increment*. This can also be done explicitly with *dch --newversion <version>*.¹

The program script uses the second option.

```
279a <AddVersionNumber 279a>≡ (280b)
function AddVersionNumber {
    # Called by DisplayDebianChangelog
    if [ -z "${Version1}" ]
    then
        RecentIdentifier
    fi
}
```

<AddVersionNumber1 281a>

First it is checked which version designation is entered in the first line of the file *debian/changelog*. It is queried whether a version designation found there is to be taken over. If the file *debian/changelog* does not exist, the version number is queried (see page 281).

```
279b <RecentIdentifier 279b>≡ (281b)
function RecentIdentifier {
    # Called by AddVersionNumber ForceOrig
    # Takes version number from debian/changelog, if it exists

    if [ -f ${GitPath}/debian/changelog ]
    then
        firstLine=$(grep --line-number 'urgency=' ${GitPath}/debian/changelog | grep '^1:')
        whiptail --title "First line:" \
            --msgbox "First line of debian/changelog;\n${firstLine}" 15 60
        recentId=$(echo ${firstLine} | sed --expression='s/^.*(//' | \
            sed --expression='s/).*//')
    fi
}
<RecentIdentifier2 280a>
```

¹New Maintainer Guide, Chap. 8.1[11]



Figure 34.2.: Display the first line of the *debian/changelog* file.

280a

<RecentIdentifier2 280a>≡

whiptail --title "Recent identifier" \
--msgbox "Recent identifier is \${recentId}" 15 60

<RecentIdentifier3 280b>

(279b)



Figure 34.3.: Recent version

280b

<RecentIdentifier3 280b>≡

if [-n "\${recentId}"]
then
Version1=\${recentId}
fi
else
InsertIdentifier
fi
}

<AddVersionNumber 279a>

(280a)

280

After that, it continues with control questions.

```

281a  <AddVersionNumber1 281a>≡ (279a)
      revisionflag=$(echo ${Version1} | grep --count '[0-9]')
      if [ ${revisionflag} -eq 0 ]
      then
          if ! whiptail --title "Identifier of the version:" \
            --defaultno --yesno "${Version1} contains no revision number.\n \
            Is it a native package?" --yes-button "Yes" --no-button "No" 15 60
          then
              InsertIdentifier
              if ! whiptail --title "Identifier of the version:" \
                --defaultno --yesno "Is ${Version1} the right identifier?" \
                --yes-button "Yes" --no-button "No" 15 60
              then
                  InsertIdentifier
              fi
          fi
      else
          if ! whiptail --title "Identifier of the version:" \
            --defaultno --yesno "Is ${Version1} the right identifier?" \
            --yes-button "Yes" --no-button "No" 15 60
          then
              InsertIdentifier
          fi
      fi
      echo "Message from AddVersionNumber: identifier=${Version1} >> ${log}
    } }

    <DebianJavabuildTemplate 246b>

```

In the *InsertIdentifier* function, first the complete identifier of the package including the version name of the revision is requested.

```

281b  <InsertIdentifier 281b>≡ (287b)
      function InsertIdentifier {
          # Called by AddVersionNumber RecentIdentifier
          RIdentifier=${Version1}
          Version1=$(whiptail --title "Identifier" \
            --inputbox "Recent identifier: ${RIdentifier}\n \
            Please insert the whole identifier of the package\n \
            (including revision version):" \
            --nocancel 15 60 3>&2 2>&1 1>&3) }

    <RecentIdentifier 279b>

```

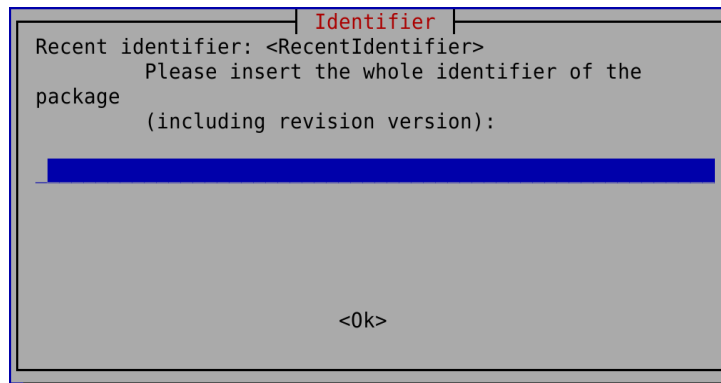


Figure 34.4.: Query the identifier

The presence of a tilde is used to check whether it is a snapshot version. If this is the case, the *gbp dch* command is also given the *--dch-opt=--force-bad-version* option and this is displayed.

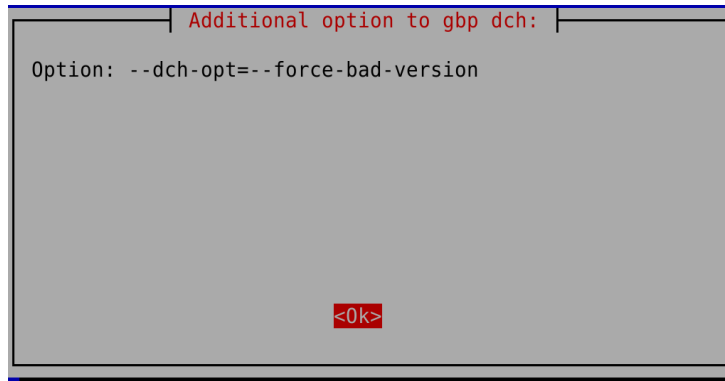
This causes the *debchange* program not to stop if the new version is smaller than the current one. This is especially useful when backporting.

```

282  <DisplayDebianChangelog3 282>≡
      SnapshotFlag=$(echo ${Version1} | grep --count '~')
      if [ ${SnapshotFlag} -eq 0 ]
      then
        DchAdd=''
      else
        DchAdd=' --dch-opt=--force-bad-version'
        whiptail --title "Additional option to gbp dch:" \
          --msgbox "Option: ${DchAdd}" 15 60
      fi
    <DisplayDebianChangelog4 283>

```

(278)

Figure 34.5.: More options for *dch*

The *debian/changelog* file is created by the program script using *gbp dch*. Always *gbp dch* is called with the options *--verbose*, *--debian-branch=* and *--new-version=*.

In case *gbp dch* fails, an error message is printed. Otherwise, the *debian/changelog* file is displayed for editing.

```

283  <DisplayDebianChangelog4 283>≡ (282)
      gbp dch --verbose --debian-branch=${RecentBranch} \
      --new-version=${Version1}${DchAdd}
      if [ $? -ne 0 ]
      then
          FailureNotice
      fi
      nano --linenumbers --mouse --softwrap debian/changelog
    fi
}

<CmeFix 249>

```

For the *tbsync* package, the following command line resulted for building a bug fix for Buster, which was too currently the stable release.

```
gbp dch --verbose --debian-branch=debian/buster
\--new-version=1.16-1-deb10u1 --dch-opt=---force-bad-version
```

The following is to be said about the background:

With `--dch-opt=<dch-option>` options for *debchange* (*dch*) can be passed to the *gbp dch* command. It should be noted that *dbp dch* calls the *dch* program multiple times and passes the option on each call. Therefore, not all *dch* options are useful at this point. Also, options may conflict with options passed by *gbp dch* by itself.

34.2. Moving the *gbp* configuration file

If a special configuration file for *git-buildpackage* is to be used for the package to be built (chapter 19.3, page 73), it is published in the directory *debian/*.

If such a configuration file was created for *gbp import-orig* when the source code was (initially) downloaded (chapter 31.4.9, page 208), it must be moved to the *debian/* directory if necessary. This is done by the *MovingGbpConfFile* function.

284a \langle *MovingGbpConf* 284a $\rangle \equiv$ (277a)
 MovingGbpConfFile
 \langle *Preparations* 285a \rangle

If a *gbp.conf* file exists in the *.git/* directory, but none exists in the *debian/* directory, this file is moved to *debian/*.

If there are corresponding files in both directories, the file to be published can be selected.

284b \langle *MovingGbpConfFile* 284b $\rangle \equiv$ (212a)
 function *MovingGbpConfFile* {
 # Called by BuildNewRevision

 # .git/gbp.conf exists, but not debian/gbp.conf
 # Move gbp.conf from .git/ to debian/
 if [-f \${GitPath}/.git/gbp.conf -a ! -f \${GitPath}/debian/gbp.conf]
 then
 mv -iv \${GitPath}/.git/gbp.conf \${GitPath}/debian
 fi
 # There is a gbp.conf in both directories
 if [-f \${GitPath}/.git/gbp.conf -a -f \${GitPath}/debian/gbp.conf]
 then
 TwoConfFilesFound
 fi
 }
 \langle *DebianBranch4Import* 205b \rangle

The function *TwoConfFilesFound* which enables this selection is described in chapter 31.4.9.

34.3. Set parameters for *gbp buildpackage*.

Before building the packages can begin, parameters for *gbp buildpackage* must first be determined and set.

```
285a  <Preparations 285a>≡ (284a)
      # Preparations for gbp buildpackage
      AskDist # Ensure that RecentBranch has a value
      <Preparations1 288a>
```

34.3.1. Identify Git branch and distribution

In order to build in the right Git branch and with the right distribution, these parameters are determined and displayed. They can also still be adjusted.

First, the *IdentifyBranches* function is used to determine the existing Git branches (chapter 29.5.2, page 152).

```
285b  <AskDist 285b>≡ (290a)
      function AskDist {
          # Called by BuildNewRevision PrepareUploading LastQuestionsBeforeBuild

          IdentifyBranches
          ba=($bl)
          for element in ${ba[*]}
          do
              # rb=$(echo ${element} | grep --count '^x_')
              # if [ $rb -ge 1 ]
              if echo ${element} | grep --quiet '^x_'
              then
                  CurrentBranch=$(echo ${element} | sed --expression='s/^x_//')
              fi
          done

          <AskDist0 285c>
```

First, *-z* is used to check whether the variable *RecentBranch* is empty. In this case the function *GitBranch2RecentBranch* (chapter 34.3.2, page 289) is called.

Otherwise, it checks if the name of the current branch corresponds to the value of the variable *RecentBranch*. If this is not the case, a hint is given and the user can select one of the two branches.

```
285c  <AskDist0 285c>≡ (285b)
      if [ -z "${RecentBranch}" ]
      then
          GitBranch2RecentBranch
      else
          if [ "${RecentBranch}" != "${CurrentBranch}" ]
          then
              Msg="Branch according to git: "${CurrentBranch}","\n \
              branch according to "${ConfigPath}${OrigName}": "${RecentBranch}
              whiptail --title "There is something wrong!" --msgbox "${Msg}" 15 60
          <AskDist1 286>
```

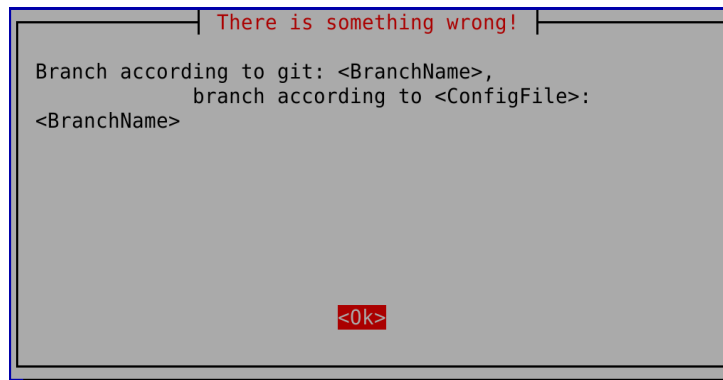


Figure 34.6.: Something is going wrong!

286 $\langle AskDist1 \ 286 \rangle \equiv$ (285c)

```

WishedBranch=$(whiptail --title "Choose branch:" \
--radiolist "Which branch do you want to work with? " \
--cancel-button "Cancel" 15 60 2 \
"0" "${RecentBranch}" off \
"1" "${CurrentBranch}" off 3>&2 2>&1 1>&3)  $\langle AskDist2 \ 287a \rangle$ 

```

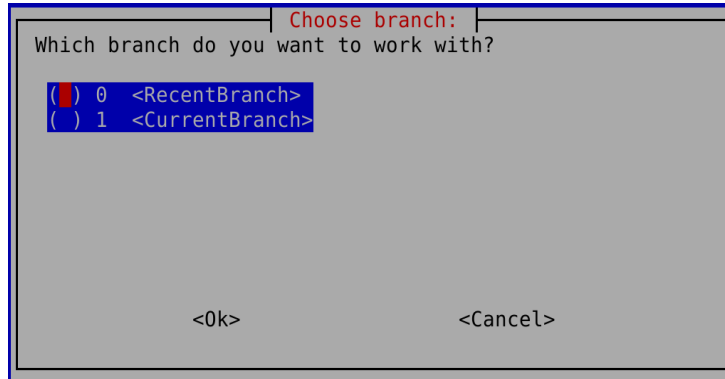


Figure 34.7.: Selection of the branch

287a $\langle AskDist2 \ 287a \rangle \equiv$ (286)

```

if [ ${WishedBranch} -eq 0 ]
then
    git checkout ${RecentBranch}
else
    GitBranch2RecentBranch
fi
fi

echo "Notice from AskDist: The branch is "${RecentBranch} >> ${log}
va=$(grep --count ${RecentBranch}_Dist ${ConfigPath}${OrigName})
if [ $va -eq 1 ]
then
    bName=${RecentBranch}
    Search4Dist
    RecentBranchD=${va}
elif [ $va -gt 1 ]
then
    nano ${ConfigPath}${OrigName}
    AskDist

```

$\langle AskDist5 \ 287b \rangle$

287b $\langle AskDist5 \ 287b \rangle \equiv$ (287a)

```

else
    Distro4Branch
fi

if [ -z "${RecentBranchD}" ]
then
    RecentBranchD="sid"
fi
echo "Notice from AskDist: The distribution is "${RecentBranchD} >> ${log}
} }

```

$\langle InsertIdentifier \ 281b \rangle$

```

288a  <Preparations1 288a>≡ (285a)
      echo "Notice from BuildNewRevision: Branch is "${RecentBranch} >> ${log}
      whiptail --title "Please check!" \
      --yesno "The git branch is "${RecentBranch}" --yes-button "Yes" \
      --no-button "No" 15 60
      rbq=$?
      <Preparations2 288b>

```

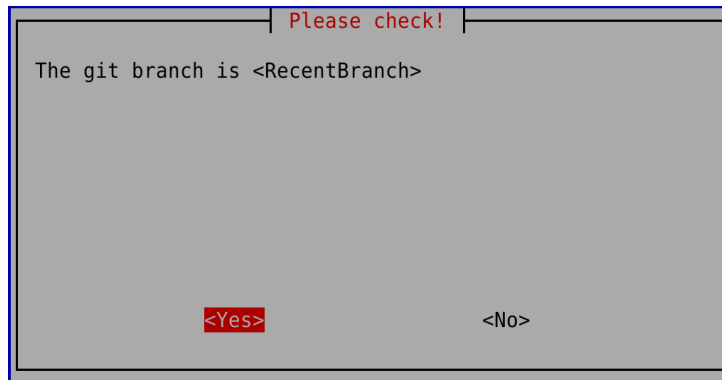


Figure 34.8.: Release-Branch

If the question whether the displayed Git branch is the correct one is answered in the negative, the applicable branch can be selected (chapter 30.4, page 163).

```

288b  <Preparations2 288b>≡ (288a)
      if [ $rbq -ne 0 ]
      then
          SelectBranch
      fi <Preparations3 288c>

```

If the question whether the displayed distribution is the correct one is answered in the negative, the applicable distribution can be selected (chapter 29.5.3, page 152).

```

288c  <Preparations3 288c>≡ (288b)
      if ! whiptail --title "Please check!" \
      --yesno "The distribution is "${RecentBranchD}" --yes-button "Yes" \
      --no-button "No" 15 60
      <Preparation4 289a>

```

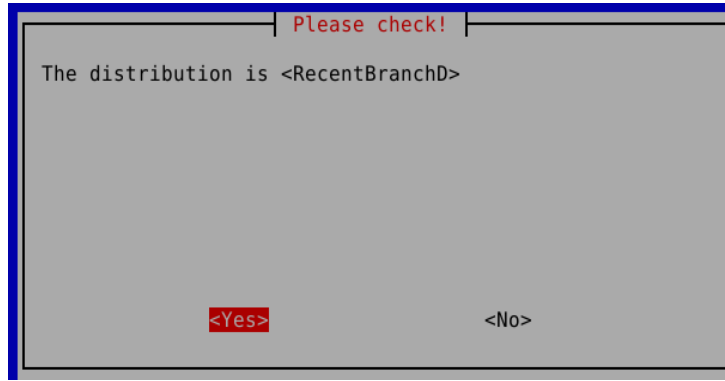


Figure 34.9.: Release branch of the distribution

```

289a  <Preparation4 289a>≡ (288c)

    then
        RecentBranchD=""
        Distro4Branch
    fi

    echo "Notice from BuildNewRevision: Distribution is "${RecentBranchD} >> ${log}

    SBuildOrPBuilder

    <BuildNewRevision8 304b>

```

34.3.2. Customize Git branch

First, in the value of the variable *CurrentBranch* (which is the name of the current branch), a "/" is replaced by "\/", that is, a slash is masked.

Then the so changed value of the variable is inserted as *RecentBranch* into the configuration file.

Finally, this value is assigned to the *RecentBranch* variable.

```

289b  <GitBranch2RecentBranch 289b>≡ (271)
    function GitBranch2RecentBranch {
        # Called by AskDist

        bName1=$(echo ${CurrentBranch} | sed --expression='s/\//\\//g')
        sed --in-place --expression="s/RecentBranch=.*\/RecentBranch=${bName1}/g" \
        ${ConfigPath}${OrigName}
        RecentBranch=${CurrentBranch}
    }

    <Search4Dist 290a>

```

34.3.3. Identify distribution

```
290a  <Search4Dist 290a>≡ (289b)
      function Search4Dist {
          # Called by AskDist ParseConfig
          va=$(grep ${bName}_Dist ${ConfigPath}${OrigName})
          bName1=$(echo ${bName} | sed --expression='s/\//\\\\/g')
          va=$(echo $va | sed --expression="s/# ${bName1}_Dist=//g")
          va=$(echo $va | sed --expression='s/"//g') }

      <AskDist 285b>
```

34.3.4. Checking the parameters

Before the package construction finally starts, the parameters are displayed for the last time for checking.

```
290b  <LastQuestionsBeforeBuild 290b>≡ (253b)
      function LastQuestionsBeforeBuild {
          # Called by UsingSBuild UsingPBuilder

          if ! whiptail --title "Please check!" \
              --yesno "The release you want to build for in ${BuildEnv} is ${RecentBranchD}" \
              --yes-button "Yes" --no-button "No" 15 60
          then
              AskDist
          fi

      <LastQuestionsBeforeBuild1 291a>
```

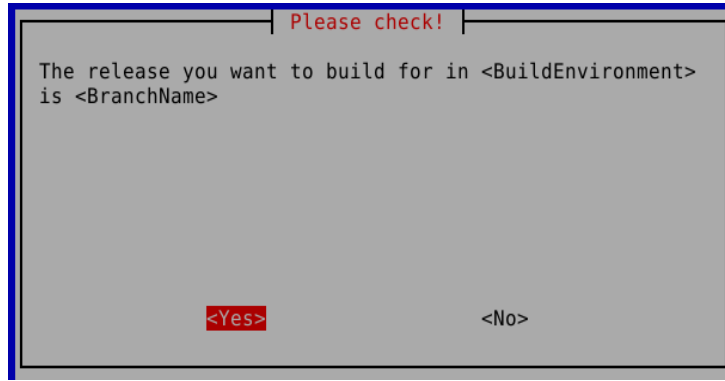


Figure 34.10.: Distribution for PBuilder

34.3.5. Last option to exit

Finally, before the package is built, a last opportunity to exit is given.

291a $\langle \text{LastQuestionsBeforeBuild1 } 291a \rangle \equiv$ (290b)
`whiptail --title "Last opportunity to exit before building" \
--yesno "Do you want to start the build process?" --yes-button "Yes" \
--no-button "Exit" 15 60`
 $\langle \text{LastQuestionsBeforeBuild2 } 291b \rangle$

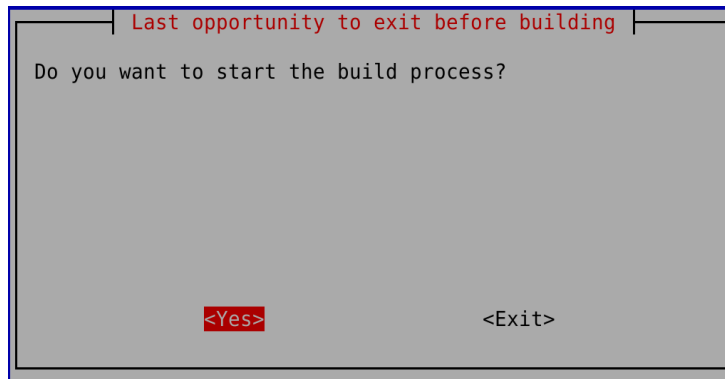


Figure 34.11.: Start building the package

291b $\langle \text{LastQuestionsBeforeBuild2 } 291b \rangle \equiv$ (291a)
`if [$? -ne 0]
then
whiptail --title "Bye" --msgbox "Exit" 15 60`
 $\langle \text{LastQuestionsBeforeBuild3 } 292 \rangle$

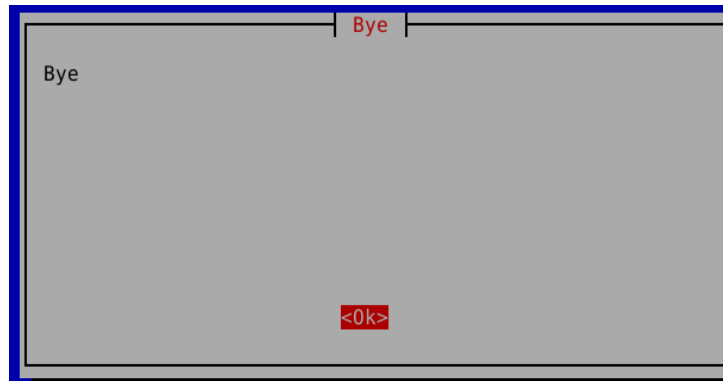


Figure 34.12.: Finish

```
292  <LastQuestionsBeforeBuild3 292>≡                                     (291b)
      exit
    else
      whiptail --title "Start building" \
        --msgbox "${BuildEnv} will be updated first\n \
        This need sudo and/or root rights" 15 60
    fi
  }

<UsingSBuild 294b>
```

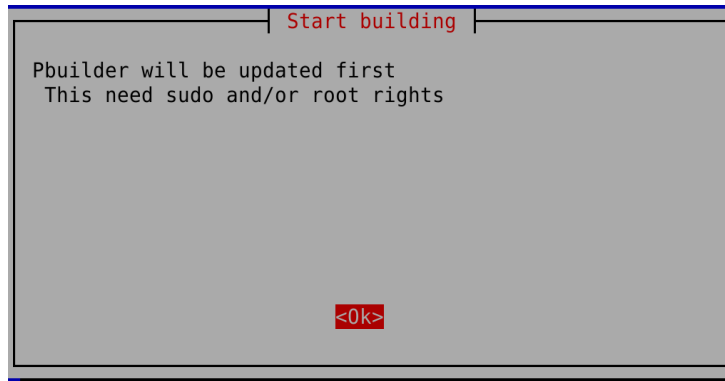



Figure 34.13.: Information about updating the build environment

34.3.6. Selecting the build system

Now you can choose whether to build with *pbuilder* or *sbuild*.

```

293  <SBuildOrPBuilder 293>≡ (304a)
      function SBuildOrPBuilder {
          # Called by BuildNewRevision TaskSelect

          Builder=$(whiptail --title "Which builder do you want to use?" \
              --radiolist "Which builder do you want to use? " 15 60 6 \
              "0" "PBuilder" off \
              "1" "SBuild" on \
              --cancel-button "Exit" 3>&2 2>&1 1>&3)
      <SBuildOrPbuilder1 294a>

```

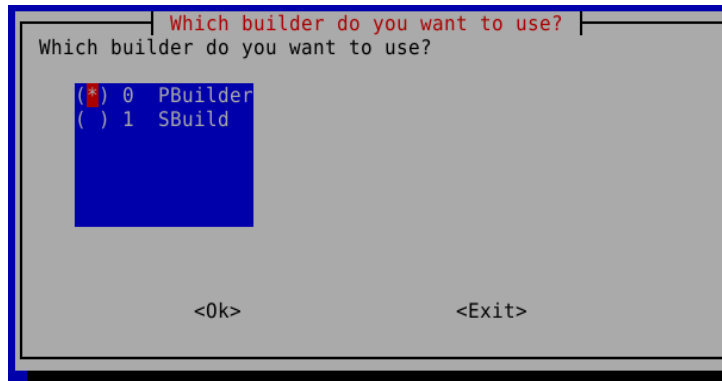


Figure 34.14.: Selecting the build system

The default is to select *sbuild*.

294a $\langle SBuildOrPbuilder1 \ 294a \rangle \equiv$ (293)

```

if [ -z "${Builder}" ]
then
    exit
fi
if [ ${Builder} -eq 1 ]
then
    echo "Using SBuild." >>${log}
    UsingSBuild
else
    echo "Using PBuilder." >>${log}
    UsingPBuilder
fi
}  $\langle BuildNewRevision \ 226 \rangle$ 

```

If *sbuild* is selected, the program script calls the *UsingSBuild* function. If *pbuilder* is selected, the *UsingPBuilder* function is called.

For a description of the *PBuilder*, see chapter 34.6 (page 297).

34.4. What does *Sbuild* do?

Sbuild is used in the official *build* network to build binary and source packages for all supported architectures.

A separate build environment is created for each.

294b $\langle UsingSBuild \ 294b \rangle \equiv$ (292)

```

function UsingSBuild {
    # Called by BuildNewRevision

    BuildEnv="sbuild"
}  $\langle UsingSBuild1 \ 295a \rangle$ 

```

First, the *LastQuestionsBeforeBuild* function is called to set the *release* parameters and the associated Git branch for *gbp buildpackage*.

```
295a <UsingSBuild1 295a>≡ (294b)
      LastQuestionsBeforeBuild

      <UsingSbuild2 297a>
```

34.5. Build in the Sbuild chroot

34.5.1. Creating the S-Chroot

First it is checked if the file *.sbuildrc* is already created in the user's home directory (chapter 25.1, page 95).

```
295b <CreateSchroot 295b>≡ (152a)
      function CreateNewSchroot {
        # Called by UsingSBuild
      , # Check whether ~/.sbuildrc exists
        if ! [ -f ~/.sbuildrc ]
        then
          # Copy from template
          cp /usr/share/doc/sbuild/examples/example.sbuildrc \
            ${HOME}/.sbuildrc

        fi

        whiptail --title ".sbuildrc (now) exists." \
          --msgbox "Please check (and edit) ~/.sbuildrc!" 15 60

      <CreateSchroot2 295c>
```

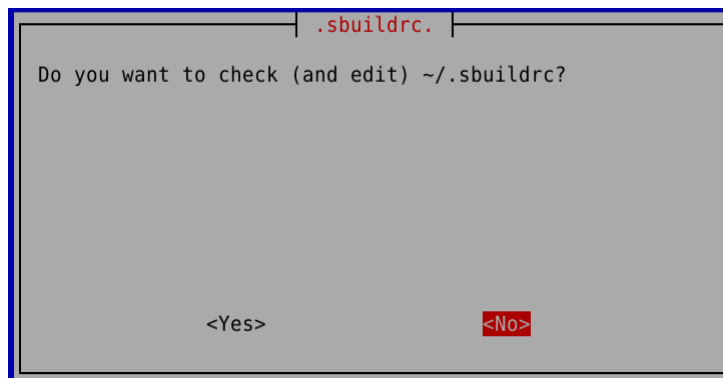


Figure 34.15.: Check *.sbuildrc*

Now the file can still be edited

```
295c <CreateSchroot2 295c>≡ (295b)
      nano ~/.sbuildrc

      <CreateSchroot3 296>
```

April 6, 2025

Then the Sbuild chroot is created.

This uses *sbuidl-createchroot* to create a *chroot* which is used by *sbuidl* to build packages for *Debian Unstable Main*.

```
296  <CreateSchroot3 296>≡ (295c)
      bDist="sid"
      bDist=$(whiptail --title "Create new Sbuild-Chroot for ${RecentBranch}" \
        --inputbox "Debian distribution\n \
        for this branch ${RecentBranch}:" \
        --cancel-button "Use Sid" 15 60 3>&2 2>&1 1>&3)
      if [ "${bDist}" = "" ]
      then
        bDist="sid"
      fi

      sudo sbuidl-createchroot ${bDist} \
        /srv/chroot/${bDist}-amd64-sbuidl \
        http://127.0.0.1:3142/deb.debian.org/debian
      echo "Schroot for "${bDist}" created." >> ${log}

  } <CreateNewCow 298a>
```

The *schroot* is placed in */srv/chroot/unstable-amd64-sbuild*. It will install the *ccache* package in the *schroot* in case you want to use some of the extensions described below. The related *apt* repository is the *http://deb.debian.org/debian* mirror service via *apt-cacher-ng*, which automatically selects a suitable local mirror. This can be changed to use a URL for a different mirror of the Debian archive.

This command can be run once per desired distribution and passed to *--arch=i386* to create a *schroot* for a different architecture (the default is the host architecture).

34.5.2. sbuild-update

The *schroot* should be up to date before building packages in it. The updates can be done with *sbuild-update*.^[SBuild2022]

```
297a  <UsingSbuild2 297a>≡ (295a)
      # Check whether chroot directory exists

      if [ -d /srv/chroot/${RecentBranchD}-amd64-sbuild ]
      then
          # if exists update-chroot
          sudo sbuild-update --update --dist-upgrade --clean --autoclean \
          --autoremove ${RecentBranchD}
          echo "Updated Schroot for ${RecentBranchD}." >>${log}
      else
          # else create chroot
          CreateNewSchroot
      fi
      # ForceOrig

      # sbuild --dist=${RecentBranchD} *.dsc
      gbp buildpackage --git-builder=sbuild \
          --git-debian-branch=${RecentBranch} \
          --git-dist=${RecentBranchD} --git-ignore-new
      gbpq=$?
```

<UsingSbuild3 297b>

All *sbuild* chroots created with *sbuild-createchroot* are created by *schroot* and have the suffix *"-sbuild"*. So, to find the names of all *sbuild* chroots, the following is executed.

```
schroot -l | grep sbuild
```

```
297b  <UsingSbuild3 297b>≡ (297a)

      build --dist=${RecentBranchD} #*.dsc
  } <UsingPBuilder 299>
```

34.6. Build in the Pbuilder chroot

The packages are built in a *chroot* provided by *pbuilder*. For better control of the process, so-called *hooks* can be built in at predefined places (chapter 18.3.3, page 65).

34.6.1. Create *base.cow*

If the required *base-cow* does not yet exist, it is created with *git-pbuilder create*. This can also be used to create a *chroot* for a different architecture such as *i386 (32-bit)* on a *64-bit* machine. For this a *base-sid-i386.cow* is created with

```
DIST=sid ARCH=i386 git-pbuilder create
```

298a (296)

```
<CreateNewCow 298a>≡
function CreateNewCow {
    # Called by Distro4Branch UsingPBuilder

    bDist="sid"
    bDist=$(whiptail --title "Create new cow for ${RecentBranch}" \
        --inputbox "Debian distribution\n \
        for this branch ${RecentBranch}:" \
        --cancel-button "Use Sid" 15 60 3>&2 2>&1 1>&3) <CreateNewCow1 298b>
```

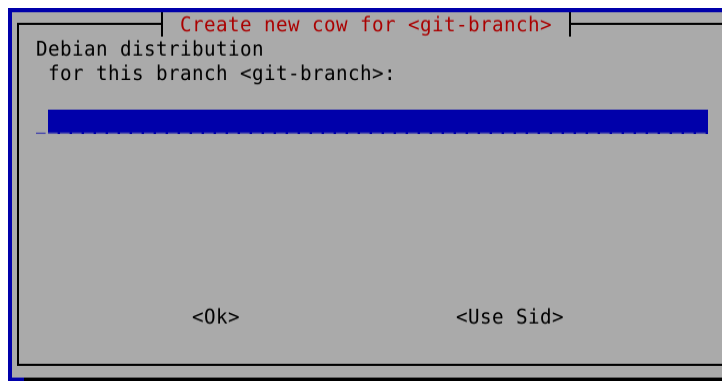


Figure 34.16.: Selection of the cow to create.

298b (298a)

```
<CreateNewCow1 298b>≡
    if [ ${bDist} == "" ]
    then
        bDist="sid"
    fi
    # You must be root to create a cow
    echo -e "\nPlease enter Password for creating pbuilder cow.\n"
    sudo DIST=${bDist} git-pbuilder create
    echo "Cow for "${bDist}" created." >> ${log}
} }
```

```
<Distro4Branch 152b>
```

With *PBuilder* an environment is created in which Debian packages can be built (see also chapter 18.3, page 61).

A wrapper is a program that surrounds another program. The wrappers mentioned below are technical in nature. *Git-pbuilder* is a wrapper for *pdebuild* and is intended for use by *gbp buildpackage*.

Git-pbuilder configures *pdebuild* to use *cowbuilder* by default.

Cowbuilder is again a *pbuilder* wrapper for *cowdancer*. *cowbuilder* performs the specified operation with *cowdancer*.

Since *git pbuilder* requires *root* privileges to update the base installation, a password prompt is issued.

34.6.2. git-pbuilder update

Before starting *gbp buildpackage*, precautions must be taken to ensure that the setup in the *chroot* is up-to-date. This is done using *git-pbuilder update*. Before this, it is still checked whether the necessary *base-cow* (of the corresponding branch/release) is available.

While the *cow* directories for the other releases have the release names in their names, the corresponding directory for *sid* is simply called *base.cow*.

Therefore, two conditions are used to check whether the corresponding directory exists.

If the corresponding directory is missing, it is created using the *CreateNewCow* function (chapter 34.6.1, page 298).

299 \langle UsingPBuilder 299 $\rangle \equiv$ (297b)

```
function UsingPBuilder {
    # Called by BuildNewRevision

    BuildEnv="pbuilder"
    LastQuestionsBeforeBuild

    # Building package using git-pbuilder and gbp buildpackage

    # check, whether cow exists
    # if exists update cow
    # else create cow
    if [ -d /var/cache/pbuilder/base- $\{RecentBranchD\}$ .cow ]
    then
        echo -e "\nPlease enter Password for updating pbuilder cow.\n"
        DIST= $\{RecentBranchD\}$  git-pbuilder update
        echo "Notice from BuildNewRevision: Pbuilder was updated." >>  $\{log\}$ 
    elif [ -d /var/cache/pbuilder/base.cow -a  $\{RecentBranchD\}$  = "sid" ]
    then
        echo -e "\nPlease enter Password for updating pbuilder cow.\n"
        DIST= $\{RecentBranchD\}$  git-pbuilder update
        echo "Notice from BuildNewRevision: Pbuilder was updated." >>  $\{log\}$ 
    else
        CreateNewCow
    fi
    ForceOrig
}
```

⟨UsingPBuilder3 303b⟩

34.6.3. Inclusion of the *.orig archive in *.changes

Basically, the *.orig archive (*.orig.tar.gz or *.orig.tar.xz) is only included in the *.changes file and thus also uploaded if the revision number does not exceed 1.

So first the version identifier is determined and displayed by calling the *RecentIdentifier* function (chapter 34.1.1, page 279).

300a ⟨ForceOrig 300a⟩≡ (231a)

```
function ForceOrig {
    # Called by BuildNewRevision PrepareUploading
    OptFlag=1
    if [ -z "${Version1}" ]
    then
        RecentIdentifier
    fi
    whiptail --title "Version" --msgbox "Version: ${Version1}" 15 60
```

⟨ForceOrig2 300b⟩

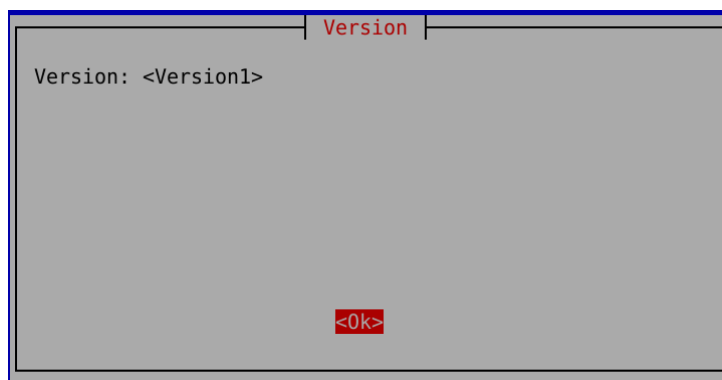


Figure 34.17.: Display of the version with revision number

From this, the revision number is now extracted and displayed. For native packages no revision number is displayed.

300b ⟨ForceOrig2 300b⟩≡ (300a)

```
cat debian/source/format | grep "native" > /dev/null
if [ $? -ne 0 ]
then
    RevNr=$(echo ${Version1} | sed --expression='s/[^0-9]/#/g' | \
    sed --expression='s/^.*#//') whiptail --title "Revision number" \
    --msgbox "The number of the revision is ${RevNr}." 15 60
    pbuilderOpt=" --git-builder=git-pbuilder \
    --git-pbuilder-options='--source-only-changes'"
```

⟨ForceOrig5 301⟩

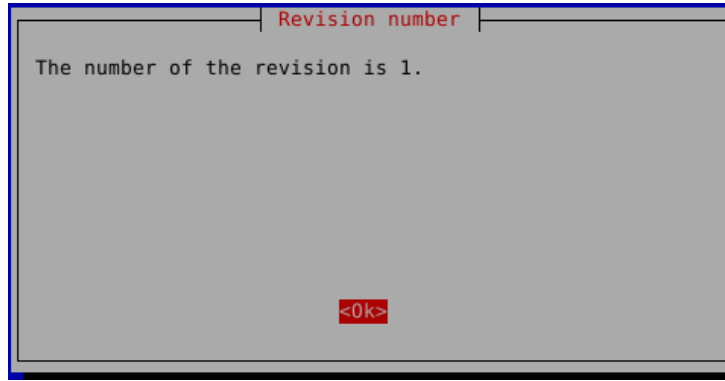


Figure 34.18.: Display of the revision number

However, you can force the *.orig archive to be included in the *.changes file and thus also uploaded if the revision number is greater than 1 by giving *git-pbuilder* the *-sa* option.

This is useful if the *.orig archive has not been uploaded yet or needs to be re-provisioned in the *New Queue*..

When using *gbp buildpackage*, as in the script, the corresponding option is *-git-builder=git-pbuilder -sa..*

```

301  <ForceOrig5 301>≡
      if [ ${RevNr} -gt 1 ]
      then
        if whiptail --title "orig in changes file" \
          --yesno "Do you want to insert the orig archive into the changes file?\n \
            That makes sense, if the orig archive has not been uploaded before." \
          --defaultno --yes-button "Yes" --no-button "No" 15 60
      <ForceOrig6 302a>

```

(300b)

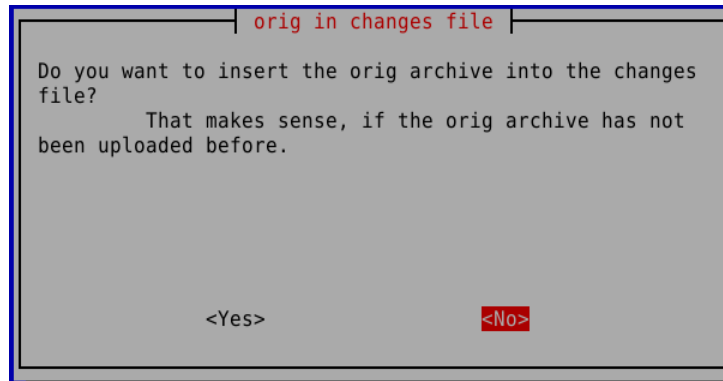


Figure 34.19.: Should the upstream tarball be uploaded too

The *-sa* option means that according to *dpkg-buildpackage --help* the source always contains *Orig*.

```
302a  <ForceOrig6 302a>≡ (301)
      then
        pbuilderOpt=" --git-builder=git-pbuilder -sa"
      fi
    fi
  fi
}
```

<MoreOptions 302b>

Now in the following function the possibility is opened to give further options for the *pbuilder* in *gbp buildpackage*. Before that the previous options are displayed.

```
302b  <MoreOptions 302b>≡ (302a)
      function MoreOptions {
        # Called by BuildNewRevision PrepareUploading
        # Adds options to specify pbuilder in gbp buildpackage
        moreOpts=''
        intText="The options for gbp buildpackage are:\n"
        intText=${intText}${normalOpts}
        if whiptail --title "Options for gbp buildpackage" \
          --yesno "${intText}\nDo you want to add some more?" --yes-button "Yes" \
          --no-button "No" --defaultno 15 60 <MoreOptions2 303a>
```

Figure 34.20.: Display the options of *gbp buildpackage*.

```

303a  <MoreOptions2 303a>≡ (302b)
      then
        moreOpts=$(whiptail --title "Options for gbp buildpackage" \
          --inputbox "${intText}\nPlease insert options to be added:" \
          --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
        moreOpts=" ${moreOpts}"
      fi
    }

```

<PatchesTreatment 251b>

The information about *pbuilderOpt* has been gathered from various manpages.

In the manpage of *gbp buildpackage* you get the information that with *-git-pbuilder-options= PBUILDER_OPTIONS* further options of *pbuilder* can be added. Which options these are can be found in the man page of *pbuilder*.

34.6.4. Build with *gbp buildpackage*

After the preparations now the building of the respective package with *gbp buildpackage* takes place. If this fails, the program terminates. Otherwise it continues to check the built packages (chapter 37, page 311). Good luck!

```

303b  <UsingPBuilder3 303b>≡ (299)

      normalOpts="--git-debian-branch="${RecentBranch}" \
        --git-dist="${RecentBranchD}" --git-verbose \
        --git-ignore-new"${pbuilderOpt}"
      MoreOptions
    <UsingPBuilder4 304a>

```

Now the actual building of the Debian package starts here with the download of the build dependencies.

```
304a  <UsingPBuilder4 304a>≡ (303b)
      echo "Starting gbp buildpackage" >> ${log}
      gbp buildpackage --git-debian-branch=${RecentBranch} \
      --git-dist=${RecentBranchD} --git-verbose \
      --git-ignore-new${pbuilderOpt}${moreOpts}
      gbpq=$?
    }
```

<SBuildOrPBuilder 293>

```
304b  <BuildNewRevision8 304b>≡ (289a)
      if [ $gbpq -eq 0 ]
      then
        echo -e "The package ${SourceName} was built with gbp buildpackage\n \
        without creating and signing tags." >> ${log}
      else
        whiptail --title "Build failed!" \
        --msgbox "Gbp buildpackage failed!" 15 60
        echo
        echo "Please fix the problem in another terminal!"
        echo "After fixing, press RETURN to continue."
        read a
        if [ "${BuildEnv}" = "sbuild" ]
        then
          UsingSBuild
        else
          UsingPBuilder
        fi
      fi
```

Task=5 # Go to RunningTests }

<RunningLintian 314a>

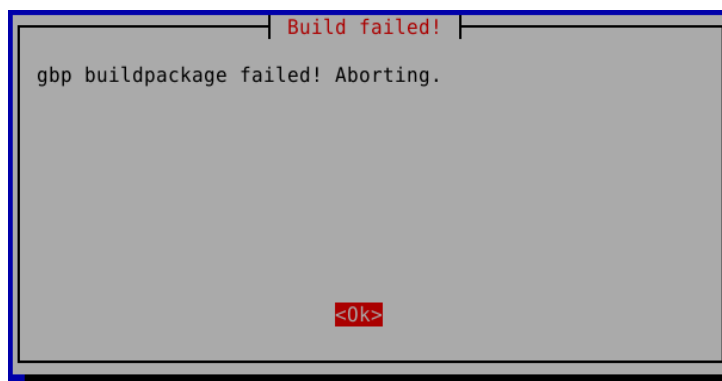


Figure 34.21.: Unsuccessful building attempt!

34.6.5. Download dependencies

34.6.6. Build - compile in *pbuilder*.

35. If building fails

If the build fails or the subsequent tests (chapter 37, page 311) are not satisfactory, this can have various causes.

First starting point for root cause investigation is to study the file
< *SourceName* >_*< Version >-< Revision >*_*< Arch >.build*.

One reason why the build failed may be insufficient determination of the build dependencies. Determining whether required dependencies are already packaged is described in chapter 10.3 (page 29).

36. Build beyond Unstable (sid)

To build for *backports* and *proposed-updates* packages (chapter 21, page 81), the following procedure has worked well for us.

First, from the output selection (chapter 30.5, page 171), a new Git branch must be created (chapter 41.1, page 355). This usually starts from the Git branch *debian/sid* or *master*.

The name of this new branch should specify the target of the package (e.g. *debian/bookworm-bpo*).

If this Git branch already exists, it is used.

309

```
<Merge2Stable 309>≡
git branch -vv
git checkout <stableBranch>
git merge debian/sid # or master
# Solve merge conflicts esp. d/changelog
nano debian/changelog
git add debian/changelog
git commit
# This is the merge commit
```

Fixing the merge conflict usually requires at least editing the *debian/changelog* file.

After that, build a new revision (chapter 32, page 225).

This creates a new entry in *d/changelog*. (Chapter 34.1, page 277)

For the version entry in *debian/changelog* it is mandatory to use nomenclature described in chapter 21.7 (page 84).

After that, the package for *proposed-update* is built. The changelog **must not** contain the number of this bug report.

If the release team approves the request, the package can be built and uploaded. (Chapter 39.1, page 334 or Chapter 40.3, page 342)

The bug report to *release.debian.org* will be closed once the package has been added to the next point release of the stable release. This is then the end of this process.

37. Verifications

An initial quality control is already performed during the construction process in *pbuilder* with *lintian*.

The following quality control is done with *lintian* in *pedantic* mode and with *uscan* regarding the content of the file *debian/watch..*

This is followed by more checks with *debdiff* and *diffoscope*.

Finally *piuparts* is described.

```
311 <RunningTests 311>≡ (318)
    function RunningTests {
        # Called by TaskSelect BuildNewRevision

        # QA using lintian and uscan

        # lintian RunningLintian

        # uscan
        if [ $!linq -eq 0 ]
        then
            RunningUscan
        else
            usq=1
        fi

        <RunningTests3 312>
```

April 6, 2025

If at least one of the two test results is qualified as poor, the program script terminates after a corresponding entry into the log file.

Otherwise there is a question whether to prepare the upload of the package.

```
312  <RunningTests3 312>≡ (311)
      if [ $usq -ne 0 ] || [ $linq -ne 0 ]
      then
        echo "At least one test failed!" >> ${log}
        exit
      else
        if whiptail --title "Upload?" \
          --yesno "Should the package be prepared to be uploaded now?" \
          --yes-button "Yes" --no-button "Exit" 15 60
        then
          Task=5 # Go to PrepareUploading
        else
          exit
        fi
      fi
}
```

<CheckRepackSuffix 223a>

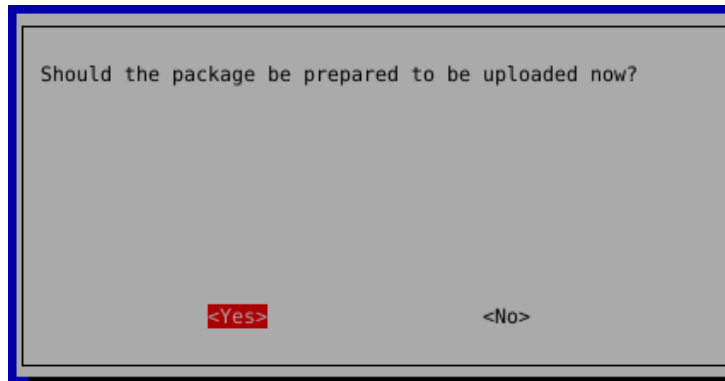


Figure 37.1.: Prepare upload of release

37.1. Selection of the Changes file

This function is used to select the changes file (**.changes*), which is used to check the package build result and determine which files to upload.

```

313 <SelectChangesFile 313>≡ (246a)
    function SelectChangesFile {
        # Called by RunningLintian SelectUploadTarget

        titlestr=${1}
        cd ${PrjPath}
        changesa=$(ls ${SourceName}_${Version1}*_*.changes)

        if [ -z "${changesa}" ]
        then
            echo "File *"${SourceName}"*_"${Version1}"*_*.changes not found!"
            exit
        fi

        i=0; slct=''
        for element in ${changesa[*]}
        do
            slct=$slct' '$i' '$element' off '
            i=$(expr $i + 1)
        done

        paket=$(whiptail --title "${titlestr}" --radiolist "Select:" \
            --cancel-button "Exit" 15 60 8 $slct 3>&2 2>&1 1>&3)

        if [ -z "${paket}" ]
        then
            exit
        fi
    }

<PatchHeader 272>

```

37.2. Yamllint

With the command line tool *yamllint* the syntax of the corresponding files can be checked.

37.3. Lintian

There is a User's Manual for *Lintian*, which is also available in the *lintian* package as file *lintian.rst* (in English)[43].

37.3.1. Test with Lintian

The result of the build is checked with *lintian*. To do this, the **.changes* file to which the check should refer must first be selected. For this the function *SelectChangesFile* is called (chapter 37.1, page 313).

```
314a  <RunningLintian 314a>≡ (304b)
      function RunningLintian {
          # Called by RunningTests

          SelectChangesFile "lintian_check" # String will be found in ${1}
          linfile=${changesa[$paket]}
      <RunningLintian1 314b>
```

Lintian is called with options that cause a verbose check.

```
314b  <RunningLintian1 314b>≡ (314a)

      lininfo=$(lintian --check --display-experimental --display-info \
      --info --verbose --show-overrides --pedantic --tag-display-limit 0 \
      --color auto ${linfile})
      lx=$?
```

<RunningLintian3 314c>

If *Lintian* does not report anything, the user should experience this pleasing result.

```
314c  <RunningLintian3 314c>≡ (314b)
      if [ ${lx} -eq 0 ]
      then
          lininfo="Lintian does not find any Errors, Warnings or \n \
          any other problems. \n\n Congratulations"
      fi
```

<RunningLintian4 315a>

The variable *lininfo* must still be edited with *sed* so that the Lintian messages appear in individual lines.

```
315a  <RunningLintian4 315a>≡ (314c)
      # Make lininfo better readable
      lininfo=$(echo ${lininfo} | sed --expression='s/ [EWIPNX]:/\n&/g')

      echo -e "lintian("${lx}"): "${lininfo}
      whiptail --title "Lintian" --msgbox "${lininfo}" --scrolltext 15 60
      echo -e "Result of Lintian:\n"${lininfo} >> ${log}
```

<RunningLintian5 315b>

After displaying the result of the test, the user must evaluate the result.

```
315b  <RunningLintian5 315b>≡ (315a)
      whiptail --title "All well?" \
      --yesno "All well? Continue?" --yes-button "Yes" \
      --no-button "Exit" 15 60
      linq=$?
  }
```

<RunningUscan 316>

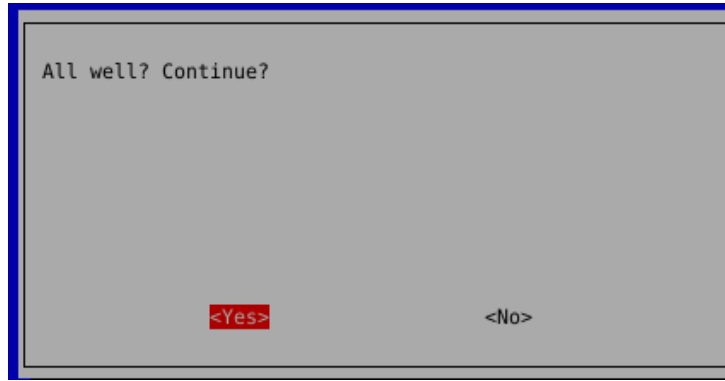


Figure 37.2.: Lintian: All Well?

If the result is OK, the next test follows.

37.3.2. Lintian reports

A complete list of possible messages can be found at <https://lintian.debian.org/tags.html>.

Here are some examples:

bad-jar-name Der Name entspricht nicht den Richtlinien der Java-Policy. ¹

codeless-jar Die *.jar Datei enthält keinen kompilierten Java-Code.

empty-binary-package Das gebaute Paket ist leer.

javalib-but-no-public-jars Im Verzeichnis */usr/share/java/* gibt es kein *.jar-Archiv.

Dann fehlt in der Regel der Eintrag in der entsprechenden Datei *debian/<paketname.poms* als *-java-lib*.

new-package-should-close-its-bug In der Datei *debian/changelog* ist der ITP-Bug zu schließen (Closes: #nnnnnn)

rules-requires-root-missing In die Datei *debian/control* wird im ersten Abschnitt der Eintrag *Rules-Requires-Root: no* benötigt.

wildcard-matches-nothing-in-dep5-copyright

backports-changes-missing

out-of-date-standard

testsuite-autopkgtest-missing

37.4. Uscan

With the option *-verbose*, *uscan* creates a human-readable report about the programme execution. With the option *-debug*, the status of the internal variables is additionally displayed.

```
316 <RunningUscan 316>≡ (315b)
    function RunningUscan {
        # Called by RunningTests BuildWithUscan
```

¹Chapter 2.4 of the Java Policy[26]


```

cd ${GitPath}
uscaninfo=$(uscan --no-download --verbose)
if [ ${#uscaninfo} -gt 0 ]
then
    whiptail --title "uscan" --msgbox "${uscaninfo}" --scrolltext 15 60
    echo -e "Result of uscan:\n"${uscaninfo} >> ${log}

```

⟨RunningUscan1 317a⟩

Here it is read from *uscaninfo* whether the built version is also the current one. This applies to all builds that are to be published in *experimental* or *sid*. The corresponding message is: "=> Package is up to date ...".

317a ⟨RunningUscan1 317a⟩≡ (316)

```

    echo ${uscaninfo} | grep '=> Package is up to date' > /dev/null
    usc1=$?
    echo ${uscaninfo} | grep '=> Only older package available' > /dev/null
    usc2=$?
    if [ ${usc1} -eq 0 ]
    then
        whiptail --title "uscan" --msgbox "Package seems to be up to date." 15 60
        usq=0

```

⟨RunningUscan2 317b⟩

317b ⟨RunningUscan2 317b⟩≡ (317a)

```

    elif [ ${usc2} -eq 0 ]
    then
        whiptail --title "uscan" --msgbox "Only older package available." 15 60
        usq=0

```

⟨RunningUscan3 317c⟩



Figure 37.3.: Older package available

317c ⟨RunningUscan3 317c⟩≡ (317b)

```

    else
        whiptail --title "No up to date message" \
        --yesno "No up to date message!\nRegardless all well? Continue?" \
        --defaultno --yes-button "Yes" --no-button "Exit" 15 60
        usq=$?
    fi

```

⟨RunningUscan4 318⟩

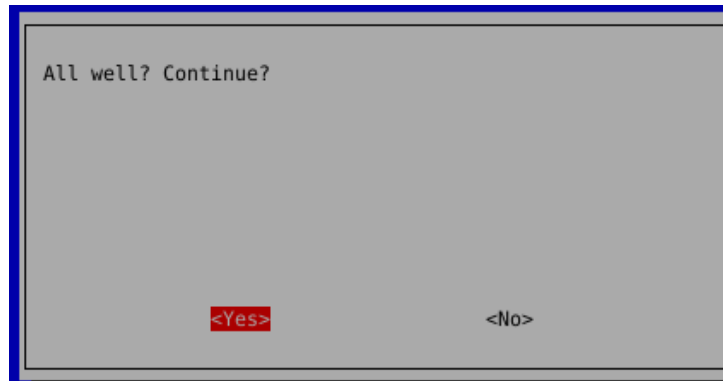


Figure 37.4.: Uscan - OK?

```
318  <RunningUscan4 318>≡ (317c)
      else
        echo "uscan failed" >> ${log}
        whiptail --title "uscan failed" --msgbox "uscan failed" 15 60
        usq=1
      fi
    }

    <RunningTests 311>
```

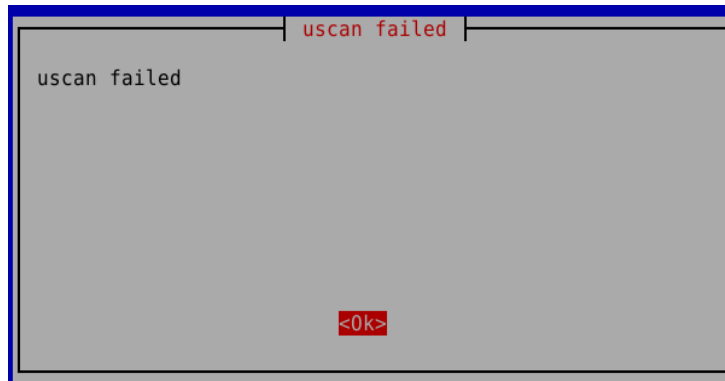


Figure 37.5.: Uscan fails

37.5. Checking the file *debian/copyright*

At this point, it is checked whether **all** details of the licences in the file *debian/copyright* have really been made. There are several tools for this. These are described in chapter 10.1 (page 27).

At present, this check still has to be done manually, because the use of these tools is sometimes insufficient.

37.6. Check with *debdiff* and *diffoscope*

Both programmes are used to show the difference between the current package and the previous version.

debdiff is installed with the *devscripts* package. *diffoscope*, on the other hand, must be installed additionally with the package of the same name.

37.6.1. *debdiff*

With *debdiff* file lists in two Debian packages can be compared.

debdiff is used in this case to prove that there are no differences between two source packets without only minor differences.

```

319 <DebDiff 319>≡
function DebDiff {
    # Called by TaskSelect
    if [ $1 -gt 0 ]
    then
        whiptail --title "debdiff" \
            --msgbox "Now you can detect the differences between two builds."\n \
            15 60
    fi
}
<DebDiff1 320>
```

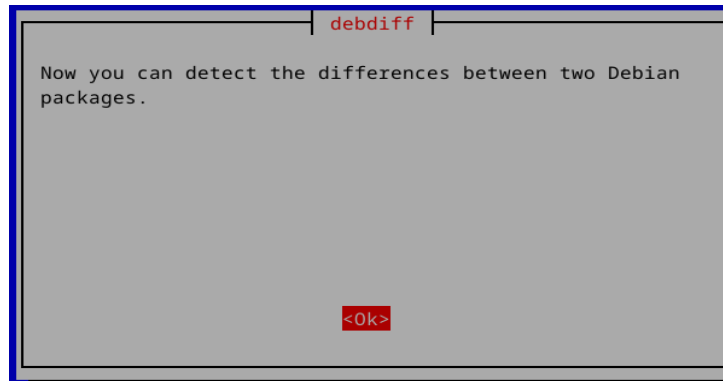


Figure 37.6.: Determine differences

For this purpose, *debdiff* is passed two source packages (.dsc files). This compares the contents of the source packages. If the source packages differ only in the Debian revision number (i.e. the .orig.tar.gz files are the same in both .dsc files), then *interdiff* is used to compare the two patch files. The *interdiff* program from the *patchutils* package must be available on the system to do this. Otherwise, a diff is performed between both source directory trees.

```

320  <DebDiff1 320>≡
      cd ${PrjPath}
      PackageList=$(ls ${PrjPath} | grep \.dsc$ | sort --reverse --version-sort)
      PackageArray=(${PackageList})

      i=0
      for element in ${PackageArray[*]}
      do
          if [ $i -eq 0 ]
          then
              i=$(expr $i + 1)
              continue
          fi
          packageE=${packageE}' '$i' '${element}' off '
          i=$(expr $i + 1)
      done

      PackageNr=$(whiptail --title "debdiff" \
          --radiolist "Which version should be compared?" 15 60 $i ${packageE} \
          --cancel-button "Cancel" \ 3>&2 2>&1 1>&3)

<DebDiff4 321>

```

321 $\langle \text{DebDiff4 } 321 \rangle \equiv$ (320)

```

if [ -z "${PackageNr}" ]
then
    CommonTasks fi

debdiff --diffstat ${PackageArray[${PackageNr}]} ${PackageArray[0]} >> \
debdiff_${PackageArray[${PackageNr}]}-${PackageArray[0]}.diff

less debdiff_${PackageArray[${PackageNr}]}-${PackageArray[0]}.diff
CommonTasks }
```

$\langle \text{ImportDebianPackage } 155a \rangle$

Part IV.

Publishing

38. Preparation to upload the package

So far, we have been busy building a **Debian** package on our machine, as also provided by the **Debian** project.

Nun geht es darum, dass dieses Paket auch hochgeladen und damit dem **Debian**-Projekt zur Verfügung gestellt werden kann.

Die Vorbereitung erfolgt durch die Funktion *PrepareUploading*.

38.1. Does *debian/changelog* exist?

To be on the safe side, it checks if a file *debian/changelog* already exists.

If this file exists, it is displayed for checking whether it is ready for publication.

```
325a <PrepareUploading 325a>≡
      function PrepareUploading {
          # Called by TaskSelect

          cd ${GitPath}

          # Check debian/changelog
          if [ -f debian/changelog ]
          then
              less --LINE-NUMBERS debian/changelog
          else
              <PrepareUploading1 325b>
```

If this file does not exist, the program script terminates at this point with a corresponding message.

```
325b <PrepareUploading1 325b>≡ (325a)
      whiptail --title "This is the end" \
      --msgbox "No changelog - no upload!" 15 60
      exit
  fi
```

```
<PrepareUploading2 326>
```



Figure 38.1.: No Changelog - No upload

If this file does not exist, the program script terminates at this point with a corresponding message.

```
326  <PrepareUploading2 326>≡ (325b)
      if ! whiptail --title "Changelog fit for publishing?" --defaultno \
      --yesno "Is the changelog fit for publishing?" --yes-button "Yes" \
      --no-button "No" 15 60
      <PrepareUploading3 327>
```

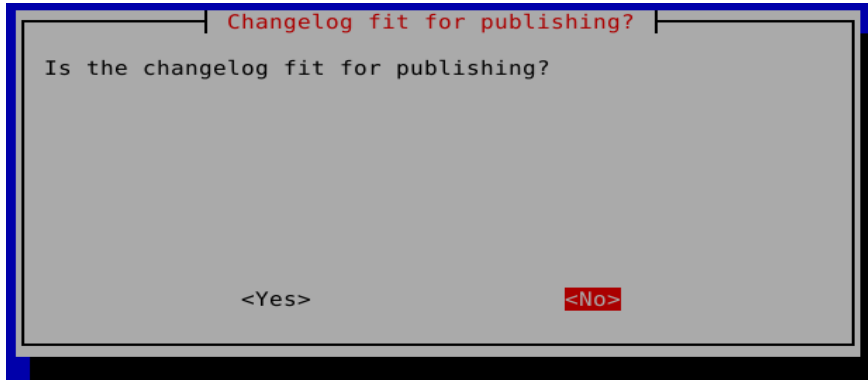


Figure 38.2.: Changelog fir for uploading?

If the answer is yes, the option to build the package again is opened (chapter 38.3, page 332).

38.2. *debian/changelog* finisg

If the *debian/changelog* file is not ready for release, it is improved.

To do this, it first prompts to check if you are in the correct `Git` branch. The *AskDist* function (chapter 34.3.1, page 285) is used to determine which is the current `Git` branch and which distribution is assigned to it.

After that, the result is displayed. If necessary, the branch can be changed in another terminal.

```
327 <PrepareUploading3 327>≡ (326)
    then
        AskDist
        echo -e "Notice from PrepareUploading: The branch is "${RecentBranch}"`n \
        The distribution is "${RecentBranchD} >> ${log}
        whiptail --title "Please check! (U)" \
        --msgbox "The branch is ${RecentBranch}" 15 60
    <PrepareUploading4 328>
```

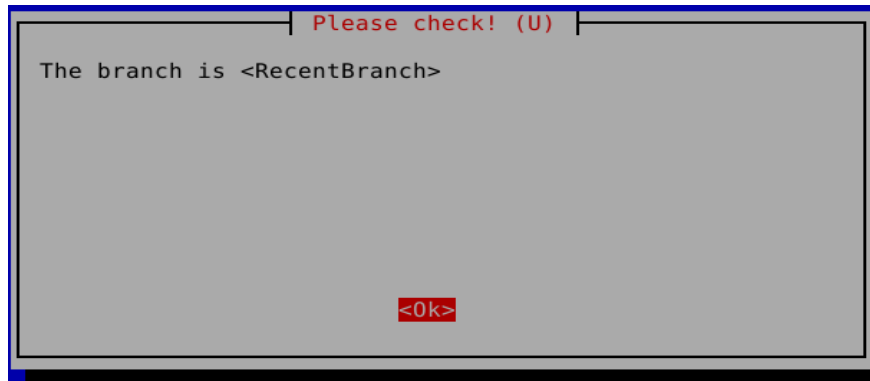


Figure 38.3.: Check branch

Then, if necessary, specify the name of the distribution where the built **Debian** package should be uploaded. Usually this is the *unstable* distribution.

328 $\langle \text{PrepareUploading}_4 \text{ 328} \rangle \equiv$ (327)

```

if [ "${RecentBranchD}" = "sid" ]
then
    distName="unstable"
elif [ "${RecentBranchD}" = "experimental" ]
then
    distName="experimental"
else
    distName=$(whiptail --title "Name of the distribution" \
        --inputbox "Please insert the name of the distribution\n \
        specified in the changelog" \
        --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
fi

```

$\langle \text{PrepareUploading}_5 \text{ 329} \rangle$

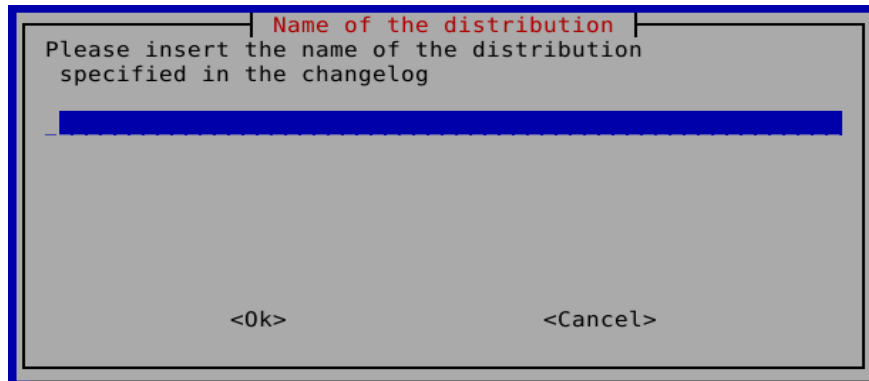


Figure 38.4.: Enter the name of the distribution

If no name is entered, *unstable* is assumed to be the distribution. .

Then the name of the distribution is displayed again with the request to check.

```
329  <PrepareUploading5 329>≡ (328)
      if [ -z "${distName}" ]
      then
          distName="unstable"
      fi

      echo -e "Another notice from PrepareUploading:\n \
The distribution is now "${distName}" >> ${log}
whiptail --title "Please check! (U)" \
--msgbox "The distribution is ${distName}" 15 60
<PrepareUploading6 330a>
```

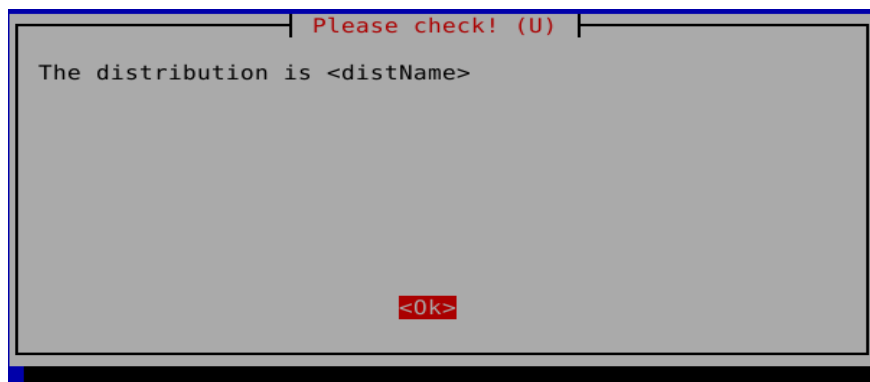


Figure 38.5.: Check distribution name

Especially for new packages, authors like to use an upload to *experimental*. This is because a *binary* upload is required for new packages.

330a $\langle \text{PrepareUploading6 } 330a \rangle \equiv$ (329)

```
# making debian/changelog fit for publishing
gbp dch --release --verbose --debian-branch=${RecentBranch} \
--distribution=${distName} #--commit
 $\langle \text{PrepareUploading8 } 330b \rangle$ 
```

After creating the changelog for the release, the standard editor is automatically opened for checking. This cannot be influenced by the script.

This display is also useful. Often there are duplicate commit entries to be deleted.

330b $\langle \text{PrepareUploading8 } 330b \rangle \equiv$ (330a)

```
git add .
git commit -a

whiptail --title "Build again" \
--msgbox "Now the release will be built another time." 15 60
```

$\langle \text{PrepareUploading10 } 331a \rangle$



Figure 38.6.: Building for release

Now the package is built for upload to the Debian repository.

To do this, the *PBuilderChroot* is updated again (chapter 34.6.2, page 299).

```
331a  <PrepareUploading10 331a>≡ (330b)
      # Building revision
      DIST=${RecentBranchD} git-pbuilder update
      GpgKeyAvailable
```

```
<PrepareUploading11 331b>
```

The first step is to make sure that the options from the *ForceOrig* function (chapter 34.6.3, page 300) are set.

```
331b  <PrepareUploading11 331b>≡ (331a)
      if [ ${OptFlag} -ne 1 ]
      then
          ForceOrig
```

```
<PrepareUploading12 331c>
```

Also, the user is given the opportunity to add more options for *gbp buildpackage*.

```
331c  <PrepareUploading12 331c>≡ (331b)
      normalOpts="--git-debian-branch=${RecentBranch}" \
      --git-dist=${RecentBranchD} --git-verbose --git-tag \
      --git-sign-tags${pbuilderOpt} MoreOptions fi
```

```
<PrepareUploading15 331d>
```

Now the actual build process takes place. This is necessary because the *debian/changelog* file has been changed and must therefore be integrated again.

```
331d  <PrepareUploading15 331d>≡ (331c)
      gbp buildpackage --git-debian-branch=${RecentBranch} \
      --git-dist=${RecentBranchD} --git-verbose --git-tag \
      --git-sign-tags${pbuilderOpt}${moreOpts}
      echo "Package ${SourceName} was built using gbp buildpackage." >> ${log}
<PrepareUploading16 332a>
```

38.3. Building again?

Here it continues if the question is answered in the affirmative whether the file *debian/changelog* is ready for release. Then the option is opened to build the package again.

Building is then done in the manner just described.

```
332a  <PrepareUploading16 332a>≡ (331d)
      else
        if whiptail --title "Building another time?" \
          --yesno "Should the release be build another time?\n(Without tagging)" \
          --yes-button "Yes" --no-button "No" 15 60
        <PrepareUploading20 332b>
```

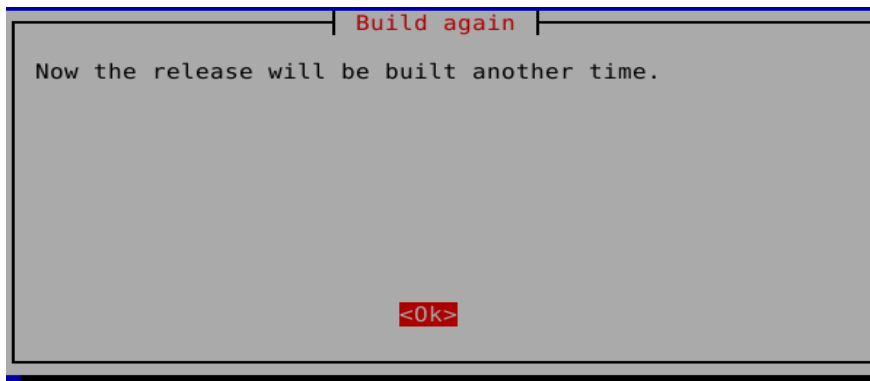


Figure 38.7.: Building for release

```
332b  <PrepareUploading20 332b>≡ (332a)
      then
        # Building revision DIST=${RecentBranchD} git-pbuilder update

        if [ ${OptFlag} -ne 1 ]
        then
          ForceOrig

          normalOpts="--git-debian-branch=${RecentBranch}" \
            --git-dist=${RecentBranchD} --git-verbose${pbuilderOpt}
          MoreOptions
        fi

        gbp buildpackage --git-debian-branch=${RecentBranch} \
          --git-dist=${RecentBranchD} --git-verbose${pbuilderOpt}${moreOpts}
        echo "Package ${SourceName} was built using gbp buildpackage." >> ${log}
      fi
    fi
  }

  <SelectUploadTarget 338>
```


39. Upload to Git repositories

The following section from the task selection first calls the functions to upload to the Git repositories.

```
333  <TaskSelect9 333>≡
      if [ ${rcts} -eq 0 ]
      then
          #####

          # Pushing git repo

          #####

          Upload2OwnServer
          Upload2Salsa

      <TaskSelect10 337a>
```

39.1. Upload to salsa.debian.org

Before uploading to *salsa.debian.org*, the program script checks whether there is already an appropriate repository there.

```

334 <Upload2Salsa 334>≡ (336)
    function Upload2Salsa {
        # Called by TaskSelect and itself

        # Uploading to Salsa

        if whiptail --title "Upload to salsa.debian.org?" \
        --yesno "Should ${SourceName} be uploaded to Salsa?" \
        --yes-button "Yes" --no-button "No" 15 60
        then
            BrowserName=$(echo ${SalsaName} | sed --expression='s/.git$//g')
            BrowserName="https://salsa.debian.org/${BrowserName}"
            wget --spider --verbose --max-redirect=0 \
            --append-output=${log} ${BrowserName}
            if [ $? -ne 0 ]
            then
                whiptail --title "No project found at salsa.debian.org" \
                --msgbox "Please create ${BrowserName} first" 15 60
                echo "No project "${BrowserName}" found at salsa.debian.org" \
                >> ${log}

                if whiptail --title "Done?" \
                --yesno "Created ${BrowserName} on salsa.debian.org?" \
                --yes-button "Yes" --no-button "No" 15 60
                then
                    Upload2Salsa
                else
                    exit
                fi
            else
                <Upload2Salsa5 335>

```

If patch queue branches exist, they can be deleted before uploading to *salsa.debian.org*.

```

335  <Upload2Salsa5 335>≡
    if echo $(git branch) | grep --quiet 'patch-queue/'
    then
        if whiptail --title "Patch queue branches found:" \
        --yesno "$(git branch | grep 'patch-queue')\n \
        Delete all patch-queue branches?" \
        --yes-button "Yes" --no-button "No" 15 60
        then
            git branch --delete --force \
            $(git branch | grep 'patch-queue')
        fi
    fi
    if ! whiptail --title "Last stop before upload!" \
    --yesno "Anything all right?" \
    --yes-button "Yes" --no-button "No" 15 60
    then
        exit
    fi
    if git remote | grep 'salsa' > /dev/null
    then
        RepoName="salsa"
    else
        RepoA=$(git remote)

        i=0; slct=''
        for element in ${RepoA[*]}
        do
            slct=$slct' '$i' '${element}' off '
            i=$(expr $i + 1)
        done

        RepoNr=$(whiptail --title "Select repository" \
        --radiolist "Select one of these repositories" \
        --cancel-button "Exit" 15 60 8 \
        $slct 3>&2 2>&1 1>&3)

        if [ -z "${RepoNr}" ]
        then
            exit
        else
            RepoName=${RepoA[${RepoNr}]}
        fi
    fi
    git push --set-upstream ${RepoName} --all >> ${log}
    git push --set-upstream ${RepoName} --tags >> ${log}

    echo "${SourceName}" was uploaded to salsa.debian.org." >> ${log}
fi
}

```

⟨*GettingFingerprint* 341⟩

39.2. Upload to the own Git-Server

Uploading to your own Git server requires that one has been set up (chapter 19.4.2, page 76). Furthermore, its name or IP address must be entered beforehand (chapter 41.2, page 356).

336 ⟨*Upload2OwnServer* 336⟩≡ (339)

```
function Upload2OwnServer {
    # Called by TaskSelect

    # Uploading to own git server
    if [ -n "$ServerName" ]
    then
        if whiptail --title "Upload to own git server?" \
        --yesno "Should ${SourceName} be uploaded to your own git server?" \
        --yes-button "Yes" --no-button "No" 15 60
        then
            git push --set-upstream home --all >> ${log}
            git push --set-upstream home --tags >> ${log}

            echo "${SourceName}" was uploaded to your git server." >> ${log}
        fi
    fi }
```

⟨*Upload2Salsa* 334⟩

40. Upload package

In the *TaskSelect* (task selection) function, the functions for uploading the packages are also called.

```
337a  <TaskSelect10 337a>≡ (333)
      #####

      # Uploading packages

      #####

      SelectUploadTarget
      CreateSignature
      UploadUsingDput
      Upload2PeopleD0
      UploadLocal
<TaskSelect11 337b>
```

If the *CreateNewBranch* (chapter 41.1, page 355), *SelectBranch* (chapter 30.4, page 163), or *OwnServer* (chapter 41.2, page 356) functions were called, the configuration file will be displayed again for editing (chapter 30.1, page 159 and then the task selection will be called again.

```
337b  <TaskSelect11 337b>≡ (337a)
      else
        ConfigFileLEC
        CommonTasks
      fi
    }

    <StartTasks (never defined)>
```

40.1. Selection of the target repository

```
338  <SelectUploadTarget 338>≡ (332b)
      function SelectUploadTarget {
          # Called by TaskSelect Upload2FtpMaster

          # Select upload target
          Upl=$(whiptail --title "Uploading?" \
            --radiolist "Should the package be uploaded to ftp-master,\n \
            people.d.o or mentors.debian.net?" 15 60 6\
            "0" "No" off \
            "1" "ftp-master" on \
            "2" "people.d.o" off \
            "3" "Mentors" off \
            "4" "Non-Maintainer-Upload" off \
            "5" "Local repository" off --cancel-button "Exit" 3>&2 2>&1 1>&3)

      <SelectUploadTarget1 339>
```

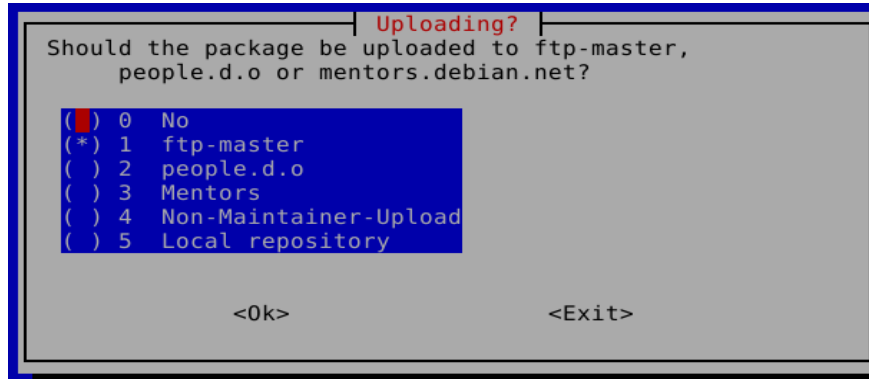


Figure 40.1.: Upload target

```

339  <SelectUploadTarget1 339>≡ (338)
    # The order of the conditions is important!
    # 'Cancel' results an empty variable
    if [ -z ${Upl} ] || [ ${Upl} -eq 0 ]
    then
        exit
    fi

    case "${Upl}" in
    1) Upltext="ftp-master";;
    2) Upltext="people.d.o";;
    3) Upltext="Mentors";;
    4) Upltext="delayed";;
    5) Upltext="local repository";;
    esac

    cd ${PrjPath}

    # Select package
    SelectChangesFile "Upload" # String will be found in ${1}
    UplPaket=${changesa[$paket]}

    Version2=$(echo ${UplPaket} | sed --expression="s/^[a-z\-\]*_//" | \
    sed --expression="s/-.*$//")
    SourceName1=$(echo ${UplPaket} | sed --expression="s/_.*$//1")
    OrigPaket=${SourceName1}_${Version2}.orig"

    # Final question before uploading starts
    if ! whiptail --title "Upload to ${Upltext}?" \
    --yesno "You want ${UplPaket} upload to ${Upltext}." \
    --yes-button "Yes" --no-button "No" 15 60
    then
        whiptail --title "Bye" --msgbox "Exit" 15 60
        exit
    fi

    echo "${UplPaket} should be uploaded to ${Upltext}." >> ${log

```

```
} }
```

```
⟨Upload2OwnServer 336⟩
```

40.2. Preparation - Create signature

The *CreateSignature* function creates the required signatures using *debsign*. *debsign* signs a Debian .changes and .dsc file pair using GnuPG. For this, the GnuPG key must be available (chapter 29.7, page 156).

In case of failure a retry is provided.

```
340  ⟨CreateSignature 340⟩≡ (341)
      function CreateSignature {
          # Called by TaskSelect Upload2FtpMaster and itself

          # Signature using debsign

          GettingFingerprint GpgKeyAvailable

          # Signature
          debsign -k${fipr} ${UplPaket}
          if [ $? -ne 0 ]
          then
              if whiptail --title "Signing failed!" \
                  --yesno "Signature failed - Retry?" \
                  --yes-button "Yes" --no-button "Exit" 15 60
              then
                  CreateSignature
              else
                  exit
              fi
          fi

          echo "${UplPaket} was signed" >> ${log}
      } }
```

```
⟨Upload2Mentors 345⟩
```


40.2.1. Use fingerprint

To sign the packages to be uploaded we need the fingerprint of the maintainer key. This is the fingerprint of the key that is also stored in the **Debian keyring**.

If there is no corresponding file in the configuration files directory, a corresponding file is created.

```

341  <GettingFingerprint 341>≡ (335)
    function GettingFingerprint {
        # Called by CreateSignature

        finchflag=0

        # getting the fingerprint of the key to sign
        if [ -f ${ConfigPath}/fingerprint ]
        then
            . ${ConfigPath}/fingerprint
        else
            mkdir --parents ${ConfigPath}
            finchflag=1
        fi

        if [ -z "${fipr}" ]
        then
            fipr=$(whiptail --title "Your fingerprint" \
                --inputbox "Please insert fingerprint of your key for signing!" \
                --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
            if [ -z "${fipr}" ]
            then
                echo "Please insert fingerprint of your key for signing!"
                read fipr
            fi
            finchflag=1
        fi

        if ! whiptail --title "Fingerprint" \
            --yesno "Is ${fipr} the right fingerprint of the key for signing?" \
            --yes-button "Yes" --no-button "No" 15 60
        then
            fipr=$(whiptail --title "Key for signing" \
                --inputbox "Real fingerprint of the key for signing:" \
                --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
            if [ -z "${fipr}" ]
            then
                echo "Please insert fingerprint of your key for signing!"
                read fipr
            fi
            finchflag=1
        fi

        if [ $finchflag -eq 1 ]
        then

```

```

        if [ -f ${ConfigPath}/fingerprint ]
        then
            mv ${ConfigPath}/fingerprint ${ConfigPath}/fingerprint.backup
            whiptail --title "Fingerprint file" \
                --msgbox "You can find the old fingerprint file as\n \
                ${ConfigPath}/fingerprint.backup" 15 60
        fi
        touch ${ConfigPath}/fingerprint
        echo "#!/usr/bin/bash" >> ${ConfigPath}/fingerprint
        echo "fipr=${fipr} >> ${ConfigPath}/fingerprint
        . ${ConfigPath}/fingerprint
    fi
}

```

⟨CreateSignature 340⟩

40.3. Upload with dput

342a *⟨UploadUsingDput 342a⟩*≡ (343)

```

function UploadUsingDput {
    # Called by TaskSelect Upload2FtpMaster

    # Uploading using dput

    cd ${PrjPath}/
    if [ ${Up1} -eq 3 ]
    then
        Upload2Mentors
    elif [ ${Up1} -eq 1 ] || [ ${Up1} -eq 4 ]
    then
        Upload2FtpMaster
    fi
}

```

⟨UploadFilesSelect 347⟩

40.4. Upload to FTP-Master

342b *⟨Upload2FtpMaster 342b⟩*≡ (344)

```

function Upload2FtpMaster {
    # Called by UploadUsingDput

    # repeat question
    if whiptail --title "Last exit" \
        --yesno "Should the package be uploaded to ftp-master?" \
        --yes-button "Yes" --no-button "No" 15 60
    ⟨Upload2FtpMaster1 343⟩
}

```

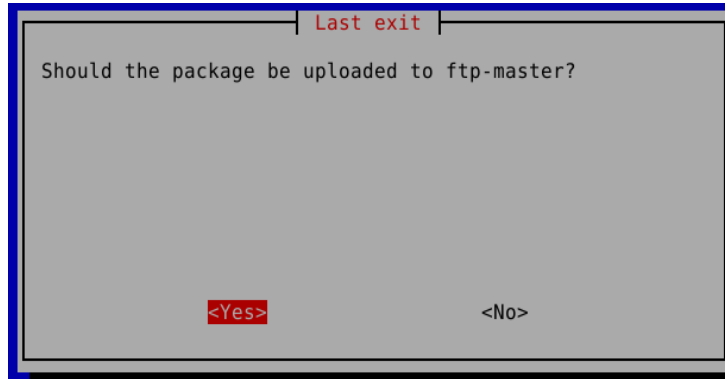


Figure 40.2.: Upload to FTP-Master - OK?

```

343  <Upload2FtpMaster1 343>≡ (342b)
      then

      # Checking whether the .changes file is the right one for the upload target
      sourceFlag=$(echo ${Up1Paket} | grep --count '_source.')
      expFlag=$(grep --line-number ' ) experimental; urgency=' ${GitPath}/debian/changelog \
      | grep '^1:')
      echo -e "${Up1Paket}:\n${expFlag}\nsourceFlag: ${sourceFlag}" >> ${log}
      # Strip line to isolate release
      expFlag=$(echo ${expFlag} | sed --expression='s/^.*) //' | \
      sed --expression='s/; .*$/')

      if [ -z "${expFlag}" ]
      then
          if [ ${sourceFlag} -eq 0 ]
          then
              if whiptail --title "Uploading?" \
              --yesno "Do you really want to upload a binary package\n \
              to ftp-master?" --yes-button "Yes" --no-button "No" 15 60
              then
                  Dput2FtpMaster
              else
                  echo "Next try to upload" >> ${log}
                  SelectUploadTarget
              fi
          else
              Dput2FtpMaster
          fi
      else
          if [ $sourceFlag -ge 1 ]
          then
              if whiptail --title "Uploading?" \
              --yesno "Do you really want to upload a source package\n \
              to experimental?" --yes-button "Yes" --no-button "No" 15 60
              then
                  Dput2FtpMaster
              else

```

```

        echo "Next try to upload" >> ${log}
        SelectUploadTarget
        CreateSignature
        UploadUsingDput
    fi
else
    Dput2FtpMaster
fi
fi
}

```

⟨UploadUsingDput 342a⟩

In the following function the upload to FTP master is done.

```

344  ⟨Dput2FtpMaster 344⟩≡ (346a)
    function Dput2FtpMaster {
        # Called by Upload2FtpMaster

        if whiptail --title "Simulate uploading?" \
        --yesno "Should the upload to ftp-master be simulated?" \
        --yes-button "Yes" --no-button "No" 15 60
        then
            dput --simulate ftp-master ${UplPaket}
            echo
            echo "After reading press return!"
            read x
        fi

        if whiptail --title "Uploading to FTP-Master?" \
        --yesno "Everything fine?\n\nShould the package be uploaded to ftp-master now?" \
        --yes-button "Yes" --no-button "No" 15 60
        then
            if [ ${Upl} -eq 1 ]
            then
                dput ftp-master ${UplPaket}
                echo "${UplPaket} was uploaded to ${Upltext}." >> ${log}
            else
                Dput2NMU
            fi
        fi
    }

```

⟨Upload2FtpMaster 342b⟩

This package initially lands on <https://incoming.debian.org/debian-builddd/pool/main/> until it is built by the build daemon (builddd) and made available for download. Provision.

40.4.1. Reject a package

When a package is rejected by the FTP masters, the subsequent upload of the corrected package should not increment the revision number.

40.5. Upload to mentors.debian.net

```

345  <Upload2Mentors 345>≡ (340)
      function Upload2Mentors {
          # Called by UploadUsingDput

          if whiptail --title "Simulate uploading?" \
          --yesno "Should the upload to Mentors be simulated?" \
          --yes-button "Yes" --no-button "No" 15 60
          then
              dput --simulate mentors ${UplPaket}
              echo
              echo "After reading press return!"
              read x
          fi

          # repeat question
          if whiptail --title "Uploading?" \
          --yesno "Should the package be uploaded to Mentors?" \
          --yes-button "Yes" --no-button "No" 15 60
          then
              dput mentors ${UplPaket}
              echo "${UplPaket} was uploade to ${Upltext}." >> ${log}
          fi
      }

      <Dput2NMU 346a>

```

40.6. Upload as *non-maintainer upload*.

In the context of fixing release-critical bugs by parties other than the package maintainer, it is common to allow the package maintainer some time to fix the problem.

```

346a  <Dput2NMU 346a>≡ (345)
      function Dput2NMU {
          # Called by Dput2FtpMaster

          DelayDays=$(whiptail --title "Non-Maintainer-Upload" \
            --radiolist "Days for delay?" 15 60 5 \
            "0" " 5 days of delay" off \
            "1" "10 days of delay" on \
            "2" "15 days of delay" off --cancel-button "Exit" 3>&2 2>&1 1>&3)

          if [ -z ${DelayDays} ]
          then
              exit
          fi

          case "${DelayDays}" in
              0) DelDays=5;;
              1) DelDays=10;;
              2) DelDays=15;;
          esac

          dput ftp-master --delayed ${DelDays} ${UplPaket}
      }

  <Dput2FtpMaster 344>

```

40.7. Upload to *people.debian.org*

```

346b  <Upload2PeopleDO 346b>≡ (347)
      function Upload2PeopleDO {
          # Called by TaskSelect

          if [ ${Upl} -eq 2 ]
          then
              # For people.d.o you can not use dput
              if whiptail --title "Archive on people.d.o" \
                --yesno "Does the directory public_html/${OrigName} \
                already exist at people.d.o?\n \
                If not you have to enter the following commands\n \
                in a separate terminal:\n\n \
                ssh <user>@people.debian.org\n \
                mkdir --parents public_html/${OrigName}" \
                --yes-button "Yes" --no-button "No" 15 60
              then
                  UploadFilesSelect
              <Upload2PeopleDO3 348>

```

347 $\langle \text{UploadFilesSelect } 347 \rangle \equiv$ (342a)

```

function UploadFilesSelect {
    # Called by Upload2PeopleDO UploadLocal
    UplFL=$(cat ${UplPaket} | grep --after-context=10 'Files: *')
    UplFL1=$(echo ${UplFL} | sed --expression='s/Files: //'')
    i=1
    while [ $i -lt 6 ]
    do
        c=$(expr ${i} '*' '5')
        UplFL2=${UplFL2}" "$(echo $UplFL1 | cut --delimiter=" " -f${c})
        i=$(expr ${i} '+' '1')
    done
}

```

$\langle \text{Upload2PeopleDO } 346b \rangle$

```

348  <Upload2PeopleDO3 348>≡ (346b)
    if whiptail --title "Upload file?" \
    --yesno "Should the following files\n${UplFL2}\n \
    have to be uploaded?" --yes-button "Yes" \
    --no-button "No" 15 60
    then
        if [ -z "${pdoaccount}" ]
        then
            pdoaccount=$(whiptail --title "Account at people.debian.org" \
            --inputbox "Please insert the name of your account on\n \
            people.debian.org" \
            --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
            if [ -z "${pdoaccount}" ]
            then
                echo "Please insert the name of your account on\n \
                people.debian.org"
                read pdoaccount
            fi
            changeflag=1
        fi

        if ! whiptail --title "Account name" \
        --yesno "The name of your account on people.debian.org:\n \
        ${pdoaccount}" --yes-button "Yes" --no-button "No" 15 60
        then
            pdoaccount=$(whiptail --title " Account name" \
            --inputbox "Name of your account on people.debian.org:" \
            --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
            if [ -z "${pdoaccount}" ]
            then
                echo "Please insert the name of your account on\n \
                people.debian.org"
                read pdoaccount
            fi
            changeflag=1
        fi

        if [ $changeflag -eq 1 ]
        then
            echo 'pdoaccount='${pdoaccount} >> ${ConfigPath}${OrigName}
            changeflag=0
        fi

        cd ${ProjectPath}/${OrigName}
        if scp -p ${UplFL2} \
        ${pdoaccount}@people.debian.org:/home/${pdoaccount}/public_html/${OrigName}
        then
            echo "${UplFL2} were uploaded to p.d.o." >> ${log}
        else
            echo "Something went wrong while uploading to p.d.o." >> ${log}
            echo "Tried to execute this command:" >> ${log}
            echo "scp -p "${UplFL2}" "${pdoaccount}"@people.debian.org:/home/"\

```



```

        ${pdoaccount}"/public_html/"${OrigName} >> ${log}
    fi
    pdoarchivetext="If the archive on people.d.o should be\n \
    used too, you have to enter the following commands:\n \
    ssh <user>@people.debian.org\n \
    cd public_html/${OrigName}\n \
    apt-ftparchive packages . > Packages\n \
    apt-ftparchive sources . > Sources\n \
    cat Packages | gzip -c > Packages.gz\n \
    cat Sources | gzip -c > Sources.gz\n \
    apt-ftparchive release . > Release"

    if whiptail --title "Archive on people.d.o" \
    --yesno "${pdoarchivetext}\n\n \
    Do you like to copy and paste these commands?" \
    --yes-button "Yes" --no-button "No" 15 60
    then
        echo -e $pdoarchivetext echo -e "\nPlease press any key to continue!"
        read x
    fi
fi
fi
}

```

⟨UpdateLocalRepo 350b⟩

40.8. Local repository

It happens again and again that packages must be built, which are needed for the actual project as dependencies. To bridge the time for going through the new-queue, these can also be provided locally for building in the *chroot*.

```
350a  <UploadLocal 350a>≡ (350b)
      function UploadLocal {
        # Called by TaskSelect
        if [ ${Up1} -eq 5 ]
        then
          # Provide for local chroot
          UploadFilesSelect
          if whiptail --title " Files Uploaded?" \
            --yesno "Should the following files\n \
            ${Up1FL2}\nhave to be uploaded?" 15 60
          then
            cd ${ProjectPath}/${OrigName}
            sudo cp ${Up1FL2} /var/local/repository
            UpdateLocalRepo
          fi
        fi
      }
```

<ChangeEntry (never defined)>

The following function already exists as a shell script under */usr/local/bin/Local-NewRepo*.

```
350b  <UpdateLocalRepo 350b>≡ (348)
      function UpdateLocalRepo {
        # Called by UploadLocal
        cd /var/local/repository

        # Make package archives writable
        # (not only for root)
        sudo chmod o+w Packages
        sudo chmod o+w Sources
        sudo chmod o+w Packages.gz
        sudo chmod o+w Sources.gz
        sudo chmod o+w Release

        # Use apt-ftparchive to update package archives sudo apt-ftparchive packages . > Packages &&

        # Reset rights
        sudo chmod o-w Packages
        sudo chmod o-w Sources
        sudo chmod o-w Packages.gz
        sudo chmod o-w Sources.gz
        sudo chmod o-w Release
      }

      <UploadLocal 350a>
```

[fuzzy]In the *chroot environment*, here */var/cache/pbuilder/base.cow*, add the following to the */etc/apt/sources.list* file:

```
deb [trusted=yes] file:///var/local/repository ./
deb-src [trusted=yes] file:///var/local/repository ./
```


Part V.

Additional components of the script

41. Another task

41.1. Create new branch

```
355 <CreateNewBranch 355>≡
function CreateNewBranch {
    # Called by TaskSelect

    # Creates a new branch (for backports or proposed-updates)
    DebianBranches
    whiptail --title "Recent branches" \
    --msgbox "Recent branches:\n${bl}" 15 60
    bName=""
    bName=$(whiptail --inputbox "Name of the new branch:" \
    --cancel-button "Cancel" 15 60 3>&2 2>&1 1>& 3)
    if [ ${bName} != "" ]
    then
        ## Create new branch in git
        git checkout -b ${bName}

        ## Change config file - make new branch to recent one
        ChangeEntry

        whiptail --title "New branch was created" \
        --msgbox "New branch ${bName} was created" 15 60
        echo "New branch ${bName} was created" >> ${log}
        Distro4Branch
    fi
}

<ParseConfig (never defined)>
```

41.2. Entering the name or IP of your own Git server

Mit dieser Funktion, die von der Aufgabenauswahl (Kapitel 30.5, Seite 171) aufgerufen werden kann, werden der Name oder die IP eines eigenen Git-Servers in die Konfigurationsdatei eingetragen. Die Eingabe eines Namens setzt eine funktionierende Namensauflösung voraus.

```

356a  [fuzzy]
      <OwnServer 356a>≡ (356b)
      function OwnServer {
          # Called by TaskSelect

          # Set name or IP of own git server
          ServerName=$(whiptail --inputbox "Name or IP-address of your git server:" \
          --cancel-button "Cancel" 15 60 3>&2 2>&1 1>&3)
          if [ -z "${ServerName}" ]
          then
              echo "Name or IP of your git server:"
              read ServerName
          fi
          if [ -n "$ServerName" ]
          then
              # ReplaceConfigLines needs two parameters:
              # name of the variable and new value
              ReplaceConfigLines 'ServerName' "${ServerName}"
              AddGitServer
          fi }

      <TaskSelect (never defined)>

```

41.3. Prov. AddGitServer

```

356b  <AddGitServer 356b>≡
      function AddGitServer {
          # Called by OwnServer
          serverlist=$(git remote -v)
          if whiptail --title "Recent remote servers" \
          --yesno "${serverlist}\nAdd git remote server 'home'?" \
          --yes-button "Yes" \ --no-button "No" 15 60
          then
              AddHomeServer
          fi
      }

      <OwnServer 356a>

```


42. Head of the Script

42.1. Shebang

[fuzzy]At the beginning of the script are the *Shebang*, notes about the authors, the version and the license.

357a `<build-gbp.sh 357a>≡
#!/usr/bin/bash`

`<copyright 357b>`

42.2. Copyright notice

he *shebang* is followed by the copyright notice.

357b `<copyright 357b>≡ (357a)
Copyright 2019-2023 Mechtilde and Michael Stehmann <mechtilde@debian.org>
version 0.8.4

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
MA 02110-1301, USA.`

`<Dependencies 358a>`

42.3. Dependencies for the program script

```
358a  <Dependencies 358a>≡ (357b)
# Dependencies: git-buildpackage, build-essential, less, pbuilder,
# pristine-tar, sudo, unzip, cowbuilder, cowdancer, debmake, quilt,
# locate, jq, lintian, devscripts, debhelper
# sbuild, schroot, debootstrap, apt-cacher-ng, devscripts
# gradle-debian-helper, maven-debian-helper, libmaven-bundle-plugin-java,
# mozilla-devscripts, zip

<Header 358b>
```

Then the dependencies are listed (chapter 18.1, page 57).

42.4. Function header

```
358b  <Header 358b>≡ (358a)
#####

# Definitions of functions

#####

<DebugRP 358c>
```

42.5. Function for troubleshooting

The following function shows a path and allows you to exit the program if necessary.

It is used for debugging and is normally unused.

```
358c  <DebugRP 358c>≡ (358b)
function DebugRP {
    # Function to show a path and give an opportunity to exit
    # It is for debugging
    descstr=${1} pathstr=${2}
    if ! whiptail --title "Shows the path" \
    --yesno "${descstr}= ${pathstr}?" --yes-button "Yes" \
    --no-button "No" 15 60
    then
        exit
    fi
}

<InsertConfigLine (never defined)>
```

Part VI.

Plugins and Scripts

43. Java-Plugin

```
361a <build-gbp-java-plugin.sh 361a>≡
#!/usr/bin/bash

# Copyright 2019-2023 Mechtilde and Michael Stehmann <mechtilde@debian.org>
# version 0.1.1

# java plugin for build-gbp.sh

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3 of the License, or
# (at your option) any later version.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
# MA 02110-1301, USA.

<Rules4Java 361b>
```

43.1. Adjustments for Java package

In the *debian/rules* file (chapter 32.4.8, page 242), the following lines are added for compiling with Java

```
361b <Rules4Java 361b>≡ (361a)
function Rules4Java {
    echo "export JAVA_TOOL_OPTIONS := -Dfile.encoding=UTF8" >> ${GitPath}/debian/rules
    echo -e "export JAVA_HOME := /usr/lib/jvm/default-java\n" >> ${GitPath}/debian/rules
} <build-gbp-java-plugin 362>
```

This means that:

1. Always use the UTF-8 encoding
2. the variable *JAVA_HOME* is set.

Only a minimal configuration of the *rules* file is provided by the script. This must often be supplemented still meaningfully. For building the JAVA packages the following options are often needed.

```
JMODS := /usr/lib/jvm/default-java/jmods
JAVACMD="\$JAVA\_HOME/bin/java"
```

It also happens that *JDK_HOME* is used instead of *JAVA_HOME*.

These additions are provided for packages built with the *maven* build system by the *Maven plugin* (chapter 44, page 363).

In the line with *dh \$@* options can be inserted.

```
--with javahelper
--buildsystem=maven
--buildsystem=ant
--buildsystem=gradle
```

362 $\langle build-gbp-java-plugin\ 362 \rangle \equiv$ (361b)

```
whiptail --title "Java plugin found" \
--msgbox "build-gbp-java-plugin.sh was loaded." 15 60

echo "build-gbp-java-plugin.sh was loaded." >> ${log}
```

44. Maven-Plugin

The **Maven** plugin supports building **Java** packages using the **Java** build system *maven* (chapter 13.4.1, page 43).

44.1. Head of the Maven plugin

The header of the plugin contains information about the creators and the license.

It also notes which **Debian** packages must be installed for the program script to run.

```
363 <build-gbp-maven-plugin.sh 363>≡
    #!/usr/bin/bash

    # Copyright 2019-2023 Mechtilde and Michael Stehmann <mechtilde@debian.org>
    # version 0.1.1

    # maven plugin for build-gbp.sh

    # This program is free software; you can redistribute it and/or modify
    # it under the terms of the GNU General Public License as published by
    # the Free Software Foundation; either version 3 of the License, or
    # (at your option) any later version.

    # This program is distributed in the hope that it will be useful,
    # but WITHOUT ANY WARRANTY; without even the implied warranty of
    # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    # GNU General Public License for more details.

    # You should have received a copy of the GNU General Public License
    # along with this program; if not, write to the Free Software
    # Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
    # MA 02110-1301, USA.

    # Dependencies: maven-debian-helper licensecheck apt-file

    <Rules4MavenDH 376>
```

44.2. Notice

The plugin script tells that it has been loaded. Thus the code of the plugin is part of the (main) program script.

```
364a  <MavenPlugin 364a>≡ (372)
      whiptail --title "maven plugin found" \
      --msgbox "build-gbp-maven-plugin.sh was loaded." 15 60

      echo "build-gbp-maven-plugin.sh was loaded." >> ${log}
      # Next the function MakeMaven is executed by the main script.
      # This is the end, my friend
```

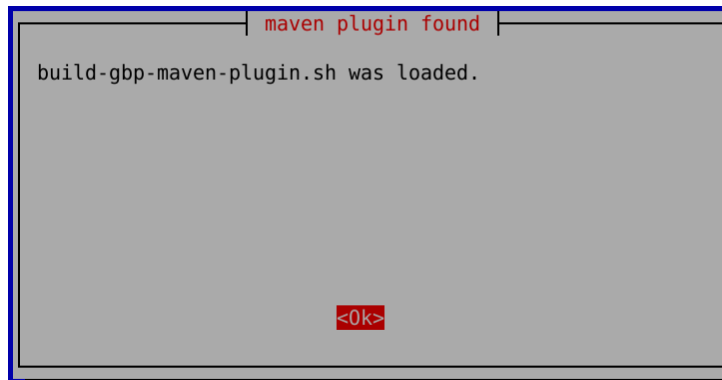


Figure 44.1.: Maven Plugin loaded

44.3. Building with Maven

This function is called by the *BuildNewVersion* function of the (main) program script after loading the plugin when building a new revision (chapter 32.1, page 226).

The execution of *mh_make* is done in a *chroot* environment. If the *chroot* does not yet exist, it must first be created as described in chapter 18.4, page 70.

```
364b  <MakeMaven 364b>≡ (369)
      function MakeMaven {
          # Called by BuildNewRevision

          cd ${GitPath}
          IdentifyMavenChrootPath

      <MakeMaven1 370>
```


365a $\langle \text{IdentifyMavenChrootPath } 365a \rangle \equiv$ (368a)

```

function IdentifyMavenChrootPath {
    # Called by MakeMaven and itself

    if [ -n "${ChrootPath}" ]
    then
        if ! whiptail --title "Maven chroot path " \
            --yesno "Is ${ChrootPath}\nthe right path to\n\
            maven chroot directory on the host?" \
            --yes-button "Yes" \
            --no-button "No" 15 60
         $\langle \text{IdentifyMavenChrootPath1 } 365b \rangle$ 

```

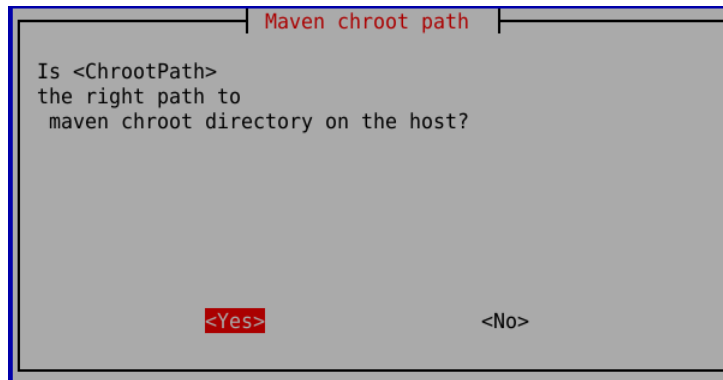


Figure 44.2.: Determine path to Maven chroot

365b $\langle \text{IdentifyMavenChrootPath1 } 365b \rangle \equiv$ (365a)

```

    then
        OldCP=${ChrootPath}
        AskChrootPath
    fi
else
    AskChrootPath
fi

 $\langle \text{IdentifyMavenChrootPath2 } 366b \rangle$ 

```

365c $\langle \text{AskChrootPath } 365c \rangle \equiv$ (373b)

```

function AskChrootPath {
    # Called by IdentifyMavenChrootPath

    # This is the way from GitPath To /srv/maven-chroot
    ChrootPath=$(whiptail --title "Maven chroot path" \
        --inputbox "Please insert the path\nto the maven chroot directory\n\
        on the host:" \ --nocancel 15 60 3>&2 2>&1 1>&3)
     $\langle \text{AskChrootPath1 } 366a \rangle$ 

```

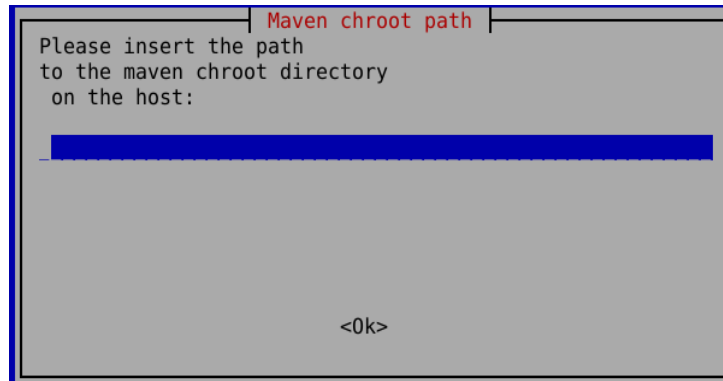


Figure 44.3.: Specify path to Maven chroot

366a $\langle AskChrootPath1 \ 366a \rangle \equiv$ (365c)

```

if [ -n "${OldCP}" ]
then
    ReplaceConfigLines "ChrootPath" "${ChrootPath}"
else
    echo "### Maven chroot path"
    echo "ChrootPath=${ChrootPath} >> ${ConfigPath}${OrigName}"
fi
OldCP=""
}

```

$\langle AskWorkspacePath \ 367c \rangle$

366b $\langle IdentifyMavenChrootPath2 \ 366b \rangle \equiv$ (365b)

```

# Create Maven-Chroot if not exists
if ! [ -d ${ChrootPath} ]
then
    echo "Please enter your SUDO password!"
    sudo mkdir --parents ${ChrootPath}
fi

if ! [ -d ${ChrootPath}/usr ]
then
    echo "Please enter your SUDO password!"
    sudo /usr/sbin/debootstrap --arch amd64 sid \
        ${ChrootPath} http://ftp.de.debian.org/debian
    echo "The maven chroot has been created." >> ${log}
fi

```

$\langle IdentifyMavenChrootPath3 \ 367a \rangle$

```

367a  <IdentifyMavenChrootPath3 367a>≡ (366b)
      # The workspace is /home/user
      if [ -n "${Path2Workspace}" ]
      then
        if ! whiptail --title "Maven chroot path " \
          --yesno "Is ${Path2Workspace}\nthe workspace in the maven chroot?" \
          --yes-button "Yes" \
          --no-button "No" 15 60
        <IdentifyMavenChrootPath4 367b>

```

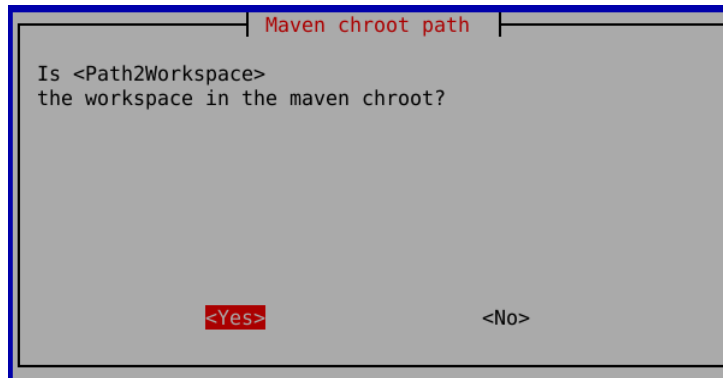


Figure 44.4.: Determine working directory in the Maven chroot

```

367b  <IdentifyMavenChrootPath4 367b>≡ (367a)
      then
        OldP2W=${Path2Workspace}
        AskWorkspacePath
      fi
    else
      AskWorkspacePath
    fi
  <IdentifyMavenChrootPath5 368b>

```

```

367c  <AskWorkspacePath 367c>≡ (366a)
      function AskWorkspacePath {
        # Called by IdentifyMavenChrootPath

        # This is the way to /home/user/
        Path2Workspace=$(whiptail --title "Maven chroot path" \
          --inputbox "Please insert the path\nto the workspace directory:" \
          --nocancel 15 60 3>&2 2>&1 1>&3)
        <AskWorkspacePath2 368a>

```

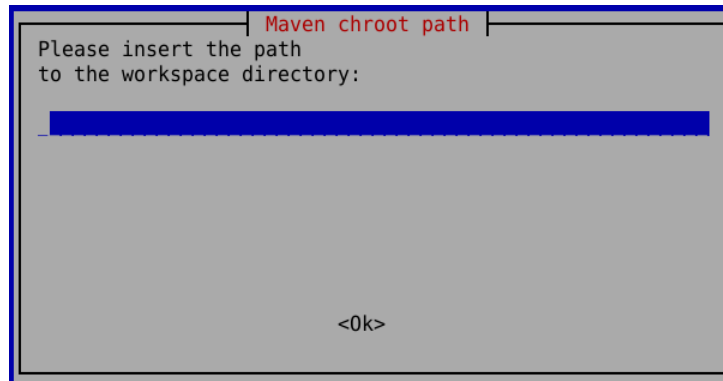


Figure 44.5.: Specify working directory in the Maven chroot

368a $\langle AskWorkspacePath2 \ 368a \rangle \equiv$ (367c)

```

if [ -n "${OldP2W}" ]
then
    ReplaceConfigLines "Path2Workspace" "${Path2Workspace}"
else
    echo "Path2Workspace=${Path2Workspace} >> ${ConfigPath}${OrigName}"
fi
OldP2W=""
}

```

$\langle IdentifyMavenChrootPath \ 365a \rangle$

368b $\langle IdentifyMavenChrootPath5 \ 368b \rangle \equiv$ (367b)

```

# Replace / at the end
Path2Workspace=$(echo ${Path2Workspace} | sed --expression="s/\$///")
MavenChrootPath=${ChrootPath}${Path2Workspace}
if ! [ -d ${MavenChrootPath} ]
then
    whiptail --title "Error!" \
        --msgbox "${MavenChrootPath} does not exist.\n\
        It will be created now." 15 60

```

$\langle IdentifyMavenChrootPath7 \ 369 \rangle$

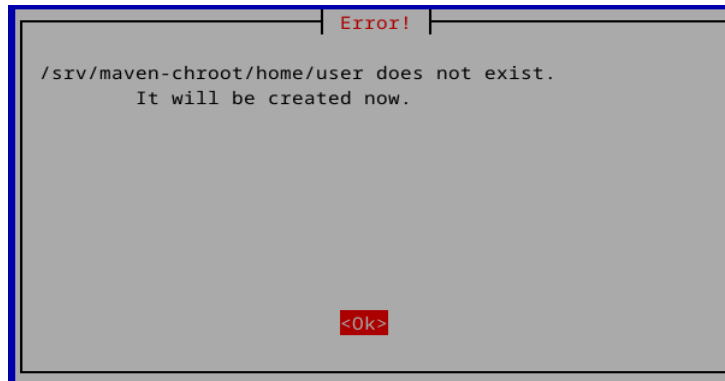


Figure 44.6.: Maven chroot does not exist

If the *chroot* does not exist, create it as described in chapter 18.4, page 70.

```

369  <IdentifyMavenChrootPath7 369>≡                                     (368b)
      echo "Please enter your SUDO password!"
      sudo mkdir --parents ${MavenChrootPath}
      echo "The maven work space has been created." >> ${log}
    fi
  }

  <MakeMaven 364b>

```

370 *⟨MakeMaven1 370⟩*≡ (364b)

```

# Copy workspace to maven chroot
sudo cp --archive ${GitPath}
${MavenChrootPath}

# Because mh_make has to run in the directory, where pom.xml is
echo "pom.xml is here:" find . -name 'pom.xml' -print
echo "Please switch to the Maven chroot in another terminal."
echo "Use these commands:"
echo
echo "# sudo mount -o bind /proc /srv/maven-chroot/proc"
echo "# sudo mount devpts /dev/pts -t devpts"
echo
echo "sudo LANG=C chroot "${ChrootPath}" /usr/bin/bash"
echo "cd "${Path2Workspace}"/"${SourceName}"
echo "try: apt install maven-debian-helper"
echo "apt update &&apt upgrade"
echo "mh_make --verbose "${SourceName}"
echo
echo "Then follow the questions of mh_make"
echo
echo "You can leave the chroot with 'exit'."
echo
echo "After finishing press return to go on!"
read a

#--run-tests=true --javadoc=true --verbose
#mh_make --package=${SourceName} --bin-package=${PackName} \
#--run-tests=false --javadoc=false --verbose
#if [ $? -ne 0 ]
#then
#    # echo "mh_make failed!"
#    # exit
#else
#    # echo "mh_make has been executed for "${PackName} >> ${log}
#fi

cp --recursive --update ${MavenChrootPath}/${SourceName}/debian \
${GitPath}

```

⟨MakeMaven5 372⟩

When running the program *mh_make* the following questions must be answered. This program itself in also a shell script.

Environment variable DEBLICENSE not set, using Apache-2.0 by default The environment variable *DEBLICENSE* is not set, then the Apache-2.0 license is used by default. In the variable *DEBLICENSE* the license for the files in *debian/* is stored.

Checking that apt-file is installed and has been configured... [ok] Check if the *apt-file* package is installed. This package is only marked as *recommended*.

Checking that licensecheck is installed... [ok] Check if the *licensecheck* package is installed. This package is also marked as *recommended* only.

Solving dependencies for package <PaketName>

Analysing pom.xml...

Resolving com.jcabi:jcabi:pom:1.21 of scope runtime...

In pom.xml: The parent POM cannot be found in the Maven repository for Debian. Ignore it? com.jcabi:jcabi:pom:1.21

That the dependency for the parent element is not found is the rule case. Therefore the question is answered in the affirmative that this can be ignored. This information is then written as *-no-parent* in the file *<packageName>.poms*.

Enter the upstream version for the package. Here the version number of the upstream code to be built is requested and usually confirmed with *Enter*. Otherwise the correct version number is entered

Version of com.jcabi:jcabi-aspects is 0.22.6 The version number is displayed again.

Choose how the version will be transformed: Now a selection is displayed how this version number should be converted.

0 - Replace all versions starting by 0. with 0.x Ersetze alle Versionsbezeichnungen

(1) - Change the version to the symbolic 'debian' version Die Standardangabe wird mit eckigen Klammern (Hier sind es aus satztechnischen Gründen runde Klammern) gekennzeichnet. Diese kann mit *Enter* bestätigt werden.

2 - Keep the version

3 - Custom rule

Andernfalls wird nun die gewünschte Ziffer eingegeben.

com.jcabi:jcabi-aspects is a bundle. - Inform mh_make that dependencies of type jar which may match this

When using *mh_make*<p0>

, note that the environment in which *mhmake* is started should have all dependencies of the package to be built. Otherwise, any entries in the relevant files (e.g. <i2>debian/control</i2>) must be made up manually. (Chapter <n0>, page <n1>)

In pom.xml: This dependency cannot be found in the Debian Maven repository. Ignore this dependency? com.jcabi:jcabi-log:0.18.1

Diese Meldung kommt, wenn die benötigte Abhängigkeit nicht auf der Arbeitsmaschine installiert ist.

```
> dpkg --search /usr/share/maven-repo/com/jcabi/jcabi-log/**
```

```
dpkg failed to execute successfully
```

```
> apt-file search /usr/share/maven-repo/com/jcabi/jcabi-log
```

```
Found /usr/share/maven-repo/com/jcabi/jcabi-log/0.18.1/jcabi-log-0.18.1.pom in libjcabi-log
```

```
Found /usr/share/maven-repo/com/jcabi/jcabi-log/0.19.0/jcabi-log-0.19.0.pom in libjcabi-log
```

```
Found /usr/share/maven-repo/com/jcabi/jcabi-log/debian/jcabi-log-debian.pom in libjcabi-log
```

April 6, 2025

```
> dpkg --search /usr/share/java/jcabi-log.jar
dpkg failed to execute successfully
> apt-file search /usr/share/java/jcabi-log.jar
Found libjcabi-log-java
[error] Package libjcabi-log-java does not contain Maven dependency
com.jcabi:jcabi-log:jar:0.17 but there seem to be a match
If the package contains already Maven artifacts but the names don't match,
try to enter a substitution rule of the form
s/groupId/newGroupId/ s/artifactId/newArtifactId/ jar s/version/newVersion/ here: >
```

After installing the missing package on the working machine, the search can be repeated.:

```
372  <MakeMaven5 372>≡ (370)
      if [ -w debian/maven.rules ]
      then
          echo "#[groupId] [artefactId] [type] [version] [classifier] [scope]" \
            >> debian/maven.rules
      fi

      ShowMaven

      # Patch for mh_make bug
      str4standardsversion="4.5.1"
      cd ${GitPath}/debian/
      less --LINE-NUMBERS control

      sed --in-place --expression=\
        "s/Standards-Version: 4.4.1/Standards-Version: ${str4standardsversion}/" \
        control
      # 'a' means append. The string after the 'a' will be appended
      # to the sting before the 'a'.
      sed --in-place \
        --expression="/Standards-Version: ${str4standardsversion}/ a Rules-Requires-Root: no" \
        control
      sed --in-place --expression=\
        "s/debhelper-compat (=12)/debhelper-compat ${str4versiondebhelpers}/" \
        control

      cd ${GitPath}
      whiptail --title "Check debian/control" \
        --msgbox "Please check the control file another time!" 15 60
        less --LINE-NUMBERS ${GitPath}/debian/control
    }

    <MavenPlugin 364a>
```


44.4. Editing Maven files

Packages built with *Maven* (*mvn*) require additional files in the *debian/* directory for this purpose. These are listed below.

373a $\langle ShowMaven\ 373a \rangle \equiv$ (376)

```
function ShowMaven {
    # Called by DisplayDebianFiles MakeMaven

    nano --linenumbers --mouse --softwrap debian/${PackName}.poms
}

 $\langle ShowMaven2\ 373b \rangle$ 

- maven.cleanIgnoreRules
- maven.properties
- maven.rules
- maven.ignoreRules
- maven.publishedRules



Now the file can still be edited


```

373b $\langle ShowMaven2\ 373b \rangle \equiv$ (373a)

```
mavenl=$(ls ${GitPath}/debian/ | grep 'maven')
for element in ${mavenl[*]}
do
    nano --linenumbers --mouse --softwrap debian/${element}
done

pomsl=$(ls ${GitPath}/debian/ | grep ${PackName})
for element in ${pomsl[*]}
do
    nano --linenumbers --mouse --softwrap debian/${element}
done
}

 $\langle AskChrootPath\ 365c \rangle$ 

```

44.4.1. maven.rules

This file is constructed as follows:

373c $\langle maven-rules.header\ 373c \rangle \equiv$

```
#[groupID] [artifactID] [type] [version] [classifier] [scope]
```

44.4.2. maven.ignoreRules

This file is constructed like the *maven.rules* file.

373d $\langle maven-ignoreRules.header\ 373d \rangle \equiv$

```
#[groupID] [artifactID] [type] [version] [classifier] [scope]
```

Artifacts that are included in the *maven.rules* file are to be deleted here if necessary.

44.4.3. maven.properties

The *maven.properties* file can be used to override values of variables within the *pom.xml* files. The most common use case is to disable or enable tests with *maven.test.skip=true*.

Other options are to change the source (*maven.compiler.source=8*) or target level (*maven.compiler.target=8*). Here the value (here: *8*) refers to the java minimum version. This entry is especially needed if a message like the following is output during compilation: *use -source 8 or higher to enable*

In addition, the file encoding used (*project.build.sourceEncoding=UTF-8*) can be set here.

```
374 <maven.properties 374>≡
    # Include here properties to pass to Maven during the build.
    # For example:
    # maven.test.skip=true # project.build.sourceEncoding=UTF-8
    # maven.compiler.source=8
    # maven.compiler.target=8
    # To skip test - missing dependencies
    maven.test.skip=true
```

44.4.4. PackageName.poms

In the file `<package name>.poms` the options to the respective `pom.xml` files are deposited.

The file then contains 2 columns. The first column contains the `pom.xml` with the corresponding path and the second column contains the corresponding options, e.g.

```
# List of POM files for the package # Format of this file is:
# <path to pom file> [option]* # where option can be:
# --ignore: ignore this POM and its artifact if any
# --ignore-pom: don't install the POM. To use on POM files that are
# created temporarily for certain artifacts such as Javadoc jars.
# [mh_install, mh_installpoms]
# --no-parent: remove the <parent> tag from the POM
# --package=<package>: an alternative package to use when installing
# this POM and its artifact
# --has-package-version: to indicate that the original version of the
# POM is the same as the upstream part of the version for the package.
# --keep-elements=<elem1,elem2>: a list of XML elements to keep in the
# POM during a clean operation with mh_cleanpom or mh_installpom
# --artifact=<path>: path to the build artifact associated with this
# POM, it will be installed when using the command mh_install. [mh_install]
# --java-lib: install the jar into /usr/share/java to comply with Debian
# packaging guidelines
# --usj-name=<name>: name to use when installing the library in /usr/share/java
# --usj-version=<version>: version to use when installing the library in
# /usr/share/java
# --no-usj-versionless: don't install the versionless link in /usr/share/java
# --dest-jar=<path>: the destination for the real jar.
# It will be installed with mh_install. [mh_install]
# --classifier=<classifier>: Optional, the classifier for the jar.
# Empty by default.
# --site-xml=<location>: Optional, the location for site.xml if it
# needsto be installed.
# Empty by default. [mh_install]
# pom.xml --no-parent --has-package-version
```

This means that the `<parent>` tag is removed from the POM when it is built. The `--has-package-version` option specifies that the original version of the POM is the same as the upstream part of the package version.

44.4.5. README.source

Information about jcabi-aspects -----

This package was debianized using the `mh_make` command from the `maven-debian-helper` package.

The build system uses Maven but prevents it from downloading anything from the Internet, making

44.5. *debian/rules* - additions for Java packages with *Maven*

In the *Rules4Maven* function, the build system specification is added to the *debian/rules* file.

```
376 <Rules4MavenDH 376>≡ (363)
    function Rules4MavenDH {
        # Called by DebianRulesTemplates

        sed --in-place \
        --expression="s/dh $@/dh $@ --buildsystem=maven" \
        ${GitPath}/debian/rules

    } <ShowMaven 373a>
```

45. Web-Extension-Plugin

The plugin shown below meets the special requirements of building Mozilla extensions as Debian packages (chapter 14, page 49).

45.1. Header for Webext-Plugins

The header of the plugin contains information about the creators and the license.

It also notes which Debian packages must be installed for the program script to run.

```
377 <build-gbp-webext-plugin.sh 377>≡
    #!/usr/bin/bash

    # Copyright 2020-2023 Mechtilde and Michael Stehmann <mechtilde@debian.org>
    # version 0.1.1

    # webext plugin for build-gbp.sh

    # This program is free software; you can redistribute it and/or modify
    # it under the terms of the GNU General Public License as published by
    # the Free Software Foundation; either version 3 of the License, or
    # (at your option) any later version.

    # This program is distributed in the hope that it will be useful,
    # but WITHOUT ANY WARRANTY; without even the implied warranty of
    # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    # GNU General Public License for more details.

    # You should have received a copy of the GNU General Public License
    # along with this program; if not, write to the Free Software
    # Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
    # MA 02110-1301, USA.

    <IdentifyWebextId 378>
```

45.2. Creating the *webext*.xpi* files in *debian/*

45.2.1. Get the name of the **.xpi* file

The extension to be packaged must be stored under the name which is composed of the designation (id) in the file *manifest.json* and the extension *.xpi*.

The designation is therefore determined as follows.

```

378 <IdentifyWebextId 378>≡ (377)
    function IdentifyWebextId {
        # Called by WebextRulesDH WebextInstall

        if [ -z ${webextID} ] &&[ -f ${GitPath}/manifest.json ]
        then
            webextID=$(grep '"id":' ${GitPath}/manifest.json)
            webextID=$(echo ${webextID} | sed --expression='s/^\"id\": \"/')
            webextID=$(echo ${webextID} | sed --expression='s/\",\\s*//')

            echo -e "Notice from Webext-Plugin: WebextID = \"${webextID}\" >> ${log}
        fi

        if [ -z ${webextID} ]
        then
            webextID="PLEASE_REPLACE_WITH_ID"
            whiptail --title "ID not found" \
                --msgbox "Didn't find ID or manifest.json in ${GitPath}" 15 60
        fi
    }

<webext-rules 379a>

```

45.2.2. *debian/rules* - Additions for Mozilla AddOns

In the *debian/rules* file for the web extension, it makes sense to list the directories or files to be installed there and assign them to a variable. This is done according to the pattern `<Package>_FILES = <List of files to include>`.

The `<package>_FILES` list is created automatically by the plugin script. It must be checked for completeness and correctness. The files to be excluded are to be entered manually.

Since the creation of the *debian/rules* file is done once, when packaging new versions, the two lists must be carefully checked and adjusted if necessary.

379a `<webext-rules 379a>≡` (378)

```
function WebextRules {
    # Called by DebianRulesTemplate

    # These strings will be added to str4rules

    Package=$(echo ${SourceName} | tr "a-z" "A-Z")
    echo -e "${Package}_FILES = \"\" >> ${GitPath}/debian/rules

    PackageL=$(ls ${GitPath})
    PackageA=(${PackageL})

    for element in ${PackageA[*]}
    do
        if [ "${element}" != "debian" ]
        then
            echo -e "${element}" \"\" >> ${GitPath}/debian/rules
        fi
    done
    echo "\$(NULL)" >> ${GitPath}/debian/rules
```

`<webext-rules5 379b>`

379b `<webext-rules5 379b>≡` (379a)

```
echo -e "\n Uncomment the following lines \n"
echo "if there are files to exclude and add them."
echo -e "\n# "${Package}"_FILES_EXCLUDE = \"\" \
>> ${GitPath}/debian/rules
echo -e "# \$(NULL)\n" >> ${GitPath}/debian/rules
}
```

`<webext-rules-dh 380>`

The following lines are added to the *str4rulesdh* variable by the program script..

```

380  <webext-rules-dh 380>≡ (379b)
      function WebextRulesDH {
        # Called by DebianRulesTemplate

        IdentifyWebextId

        DHCleanStr="override_dh_clean:\n\tdh_clean\n\trm -rf debian/build\n"

        DHAutoBuildIndepStr1="override_dh_auto_build-indep:"
        DHAutoBuildIndepStr2="\tmkdir \$(CURDIR)/debian/build &&\\"
        Str3B="\tzip --recurse-paths "
        DHAutoBuildIndepStr3="\${Str3B}"\$(CURDIR)/debian/build/"\${webextID} ".xpi \\"
        DHAutoBuildIndepStr4="\t \${Package}_FILES) \\"
        DHAutoBuildIndepStr5="# \t --exclude \${Package}_FILES_EXCLUDE)"
        DHAutoBuildIndepStr6="\tdh_auto_build\n"

        echo -e \${DHCleanStr} >> \${GitPath}/debian/rules
        echo -e \${DHAutoBuildIndepStr1} >> \${GitPath}/debian/rules
        echo -e \${DHAutoBuildIndepStr2} >> \${GitPath}/debian/rules
        echo -e \${DHAutoBuildIndepStr3} >> \${GitPath}/debian/rules
        echo -e "\${DHAutoBuildIndepStr4}" >> \${GitPath}/debian/rules
        echo -e "\${DHAutoBuildIndepStr5}" >> \${GitPath}/debian/rules
        echo -e \${DHAutoBuildIndepStr6} >> \${GitPath}/debian/rules
      }

      <WebextControl 381a>

```


45.2.3. *debian/control* - Additions for Mozilla AddOns

```

381a  <WebextControl 381a>≡ (380)
      function WebextControl {
          # Called by DebianControlTemplate
          # TB specific, for FF 'web' sed --in-place \
          --expression="s/Section: /Section: mail/" ${GitPath}/debian/control

          # 'a' means append. The string after the 'a' will be appended
          # to the sting before the 'a'.
          # @X escapes the space at the beginning of the appended line.
          # It will be removed later.
          sed --in-place \
          --expression="/Build-Depends: debhelper-compat ${str4versiondebhelpers}/ \
          a @X , zip" \
          ${GitPath}/debian/control
          # TB specific, for FF 'firefox-esr (>= 91.4)' sed --in-place \
          --expression="/Depends: \${misc:Depends}/ \
          a @X , thunderbird (>= 1:91.4)" \
          ${GitPath}/debian/control
          sed --in-place --expression="s/^@X//g" ${GitPath}/debian/control
      }

      <WebextInstall 381b>

```

45.2.4. *debian/webext-*.install*

For Mozilla extensions the following entry is mandatory:

```

debian/build/<manifest-id>.xpi /usr/share/webext

```

```

381b  <WebextInstall 381b>≡ (381a)
      function WebextInstall {
          # Called by DisplayDebianFiles
          IdentifyWebextId
          InstallStr="debian/build/"${webextID} ".xpi\tusr/share/webext"
          echo -e ${InstallStr} >> ${GitPath}/debian/${PackName}.install
      }

      <WebextDocs 381c>

```

45.2.5. *debian/webext-*.docs*

```

381c  <WebextDocs 381c>≡ (381b)
      function WebextDocs {
          # Called by echo "Still empty"
      }

      <WebextLinks 382a>

```

45.2.6. *debian/webext-links-tb*

```
\# Source Target  
/usr/share/webext/<manifest.id>.xpi /usr/lib/thunderbird/extensions/<manifest.id>.xpi
```

382a \langle *WebextLinks* 382a $\rangle \equiv$ (381c)

```
function WebextLinksTB {  
    # Called by DisplayDebianFiles  
    IdentifyWebextId  
    SourceStr="/usr/share/webext/"${webextID}".xpi\t"  
    TargetStr="/usr/lib/thunderbird/extensions/"${webextID}".xpi"  
    echo -e ${SourceStr}${TargetStr} >> \  
    ${GitPath}/debian/${PackName}.links  
}
```

\langle *webext-plugin-end* 382b \rangle

Now comes the conclusion of this Webext plugin.

382b \langle *webext-plugin-end* 382b $\rangle \equiv$ (382a)

```
whiptail --title "webext plugin found" \  
--msgbox "build-gbp-webext-plugin.sh was loaded." 15 60  
# This is the end, my friend
```

46. Python-Plugin

The plugin shown below meets the special requirements of building **packages** written in the **Python** programming language. (Chapter 15, page 51).

```
383a <build-gbp-python-plugin.sh 383a>≡
    #!/usr/bin/bash

    # Copyright 2020-2022 Mechtilde and Michael Stehmann <mechtilde@debian.org>
    # version 0.1.1

    # python plugin for build-gbp.sh

    # This program is free software; you can redistribute it and/or modify
    # it under the terms of the GNU General Public License as published by
    # the Free Software Foundation; either version 3 of the License, or
    # (at your option) any later version.

    # This program is distributed in the hope that it will be useful,
    # but WITHOUT ANY WARRANTY; without even the implied warranty of
    # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    # GNU General Public License for more details.

    # You should have received a copy of the GNU General Public License
    # along with this program; if not, write to the Free Software
    # Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
    # MA 02110-1301, USA.
```

<python-rulesDH 384b>

Next, the plugin announces that it has been loaded.

```
383b <IdentifyPythonId 383b>≡ (385a)
    whiptail -title "python plugin found" \
    -msgbox "build-gbp-python-plugin.sh was loaded." 15 60

    echo "build-gbp-python-plugin.sh was loaded." >> ${log}

    <python-plugin-end 385b>
```

46.1. Customizations for Python packages

Here, the specifics for Python packaging are mapped to the *debian/rules* file.

```
384a  <python-rules 384a>≡ (384b)
      function PythonRules {
          # Called by DebianRulesTemplate

          # These strings will be added to str4rules
          echo -e "export PYBUILD_NAME=\"${SourceName}\"\\n" >> ${GitPath}/Debian/rules
          echo -e "export PYBUILD_SYSTEM=distutils\\n" >> ${GitPath}/Debian/rules
      }
```

<python-control 384c>

```
384b  <python-rulesDH 384b>≡ (383a)
      function PythonRulesDH {
          # Called by DebianRulesTemplate

          sed --in-place \
            --expression="s/dh $@/dh $@ -with python3 --buildsystem=pybuild/" \
            ${GitPath}/debian/rules

      } <python-rules 384a>
```

46.2. *debian/control* - Addition for Python packages

Here, the specifics for Python packaging are mapped to the *debian/control* file.

```
384c  <python-control 384c>≡ (384a)
      function PythonControl {
          # Called by DebianControlTemplate

          # Recent Python version
          PythonVersion=$(py3versions --installed)

          sed --in-place \
            --expression="s/Section:/Section: python/" ${GitPath}/debian/control

          sed --in-place \
            --expression="/Build-Depends: debhelper-compat ${str4versiondebhelpers}/ \
            a , dh-python@X , python3-all@X , python3-setuptools@X\
            X-Python3-Version: >=${PythonVersion}" ${GitPath}/debian/control

          sed --in-place \
            --expression=""
```

<python-control1 385a>

For testing, the *python3-distutils* and *python3-distutils-extra* packages are also required as build dependencies.

385a $\langle python-control1$ 385a \equiv (384c)

```
sed --in-place \
--expression="/Depends: \${misc:Depends}/ \
a @X , \${python3:Depends}" \
\${GitPath}/debian/control

sed --in-place --expression="s/@X/\n/g" \${GitPath}/debian/control
}
```

$\langle IdentifyPythonId$ 383b \rangle

385b $\langle python-plugin-end$ 385b \equiv (383b)

```
# This is the end, my friend
```


47. Scripts

47.1. Creation of a project within the Java team

Creating a project in the Java team, including obtaining an access token to be assigned to the *SALSA_TOKEN* variable in the following script, is described in chapter ?? (page ??).

```
387a <setup-salsa-repository 387a>≡
#!/usr/bin/bash
#
# Setup a new Git repository on Salsa
#
# This script uses the GitLab REST API and requires an access token.
# The token is obtained from the GitLab profile page -> Access Tokens
# (https://salsa.debian.org/profile/personal_access_tokens).
# The token is in the environment variable SALSA_TOKEN
# or has to be sourced from the ~/.salsarc file and assigned
# to the SALSA_TOKEN variable:
#
# This is the Token for jollyday-java
# SALSA_TOKEN="KyuzRyWTTxfddcGcyphN" #

set -eu
<setup-salsa-repository3 387b>
```

The *jq* package must be installed on the local machine.

```
387b <setup-salsa-repository3 387b>≡ (387a)

if ! which jq >/dev/null
then
    echo "You need to apt install jq" >&2
    exit 1
fi

<setup-salsa-repository5 388a>
```

April 6, 2025

The call is made with the specification of the *SourceName*.

388a `<setup-salsa-repository5 388a>≡` (387b)

```
if [ -z "$1" ];
then
    echo "Usage: ./setup-salsa-repository <packagename>"
    exit 1;
fi

check_return_code() {
    if [ $? -ne 0 ];
    then
        echo
        echo "Something went wrong!"
        exit 1
    fi
}
```

```
test -n "$SALSA_TOKEN" || . ~/.salsarc
```

```
PACKAGE=$1
```

```
SALSA_URL="https://salsa.debian.org/api/v4"
```

```
SALSA_GROUP=java-team
```

```
SALSA_GROUP_ID=2588
```

`<setup-salsa-repository5 388a>`

Now the repository is created on *salsa.debian.org*.

389 `<setup-salsa-repository8 389>≡` (388a)

```
# -----
```

```
echo "Creating the ${PACKAGE} repository..."
```

```
RESPONSE=$(curl -s "$SALSA_URL/projects?private_token=$SALSA_TOKEN" \
```

```
--data "path=$PACKAGE&namespace_id=\ $SALSA_GROUP_ID&visibility=public&issues_enabled=false&snippe
```

```
echo $RESPONSE | jq --exit-status .id > /dev/null
```

```
check_return_code
```

```
PROJECT_ID=$(echo $RESPONSE | jq '.id')
```

`<setup-salsa-repository10 ??>`

Now the pending BTS tag hook is configured.

```
389 <setup-salsa-repository8 389>≡ (388a)
# -----

echo "Configuring the BTS tag pending hook..."

TAGPENDING_URL="https://webhook.salsa.debian.org/tagpending/$PACKAGE"
curl --silent --output /dev/null -XPOST --header "PRIVATE-TOKEN: $SALSA_TOKEN" \
$SALSA_URL/projects/$PROJECT_ID/hooks \
--data "url=$TAGPENDING_URL&push_events=1&enable_ssl_verification=1"
check_return_code

# -----

echo "Configuring the KGB hook..."

KGB_URL="http://kgb.debian.net:9418/webhook/?channel=debian-java\

%26network=oftc%26private=1%26use_color=1%26use_irc_notices=1%26squash_threshold=20"
curl --silent --output /dev/null -XPOST --header "PRIVATE-TOKEN: $SALSA_TOKEN" \
$SALSA_URL/projects/$PROJECT_ID/hooks \
--data "url=\
$KGB_URL&push_events=yes&issues_events=yes&merge_requests_events=
yes&tag_push_events=\ yes&note_events=yes&job_events=yes&pipeline_events=yes&wiki_events=yes&enab

check_return_code

# -----

echo "Configuring email notification on push..."

curl --silent --output /dev/null -XPUT --header "PRIVATE-TOKEN: $SALSA_TOKEN" \
$SALSA_URL/projects/$PROJECT_ID/services/emails-on-push \
--data "recipients=pkg-java-commits@lists.alioth.debian.org \
dispatch@tracker.debian.org"
check_return_code

# -----

echo
echo "Done! The repository is located at ${SALSA_URL
```

47.2. Script for extracting the documentation in PDF and Epub format.

This script is used to create readable documents.

47.2.1. Dependencies

The following packages must be installed for the script to run.

- noweb
- texlive
- texlive-latex-extra
- texlive-extra-utils
- texlive-binaries
- texlive-bibtex-extra
- biber
- texlive-lang-japanese
- tidy
- texlive-lang-german - for the German documentation
- texlive-lang-english - for the English translation
- texlive-lang-french - for the French translation

47.2.2. Procedure

First, the script creates *.tex documents from the *.nw documents.

The following script can be extracted with

```
notangle -Rcreate-book.sh Part6.nw > create-book.sh &&t=$(date +
echo "#generated on $t" >> create-book.sh
chmod ugo+x create-book.sh
```

from the file *Part6.nw*.

```
390 <create-book.sh 390>≡
    #!/usr/bin/bash

    #set -e

    BasePath=$(pwd)

    LANGS="en_US"

    noweave -index -delay BuildWithGBP.nw > BuildWithGBP.tex # contains preamble
    noweave -index -delay Title.nw > GBP-Title.tex
    noweave -index -delay Part1.nw > GBP-Part1.tex
    noweave -index -delay Part2.nw > GBP-Part2.tex
    noweave -index -delay Part3.nw > GBP-Part3.tex
    noweave -index -delay Part4.nw > GBP-Part4.tex
    noweave -index -delay Part5.nw > GBP-Part5.tex
```

```
noweave -index -delay Part6.nw > GBP-Part6.tex
```

```
#noweave -filter l2h -index -html BuildWithGBP.nw | htmlltoc > BuildWithGBP.html
```

<create-book.sh1 391a>

This will then be converted to *.pdf and *.epub documents. The keyword directory must be created only once, after the first execution of *pdflatex*. Multiple execution generates too high page numbers.

391a *<create-book.sh1 391a>*≡ (390)

```
if [ -f BuildWithGBP.aux ]
then
    rm BuildWithGBP.aux
fi

for ((i=4; i>0; i--))
do
    echo $i
    pdflatex -shell-escape BuildWithGBP.tex

    # Create Index only one time
    # Otherwise you get wrong page numbers
    if [ i=4 ]
    then
        makeindex BuildWithGBP
    fi

    # Create Bibliography
    biber BuildWithGBP
done
```

<create-book.sh5 391b>

The EPUB format is a zipped XHTML enriched with metadata.

391b *<create-book.sh5 391b>*≡ (391a)

```
# tex4ebook -f mobi BuildWithGBP.tex
tex4ebook -f epub BuildWithGBP.tex ../
```

<create-book.sh6 392>

392 $\langle \text{create-book.sh6 } 392 \rangle \equiv$

(391b)

```
# For the first translation

for lang in ${LANGS}
do
    cd translation/${lang}/target
    for ((i=4; i>0; i--))
    do
        pdflatex -shell-escape ./BuildWithGBP.tex
        if [ i=4 ]
        then
            makeindex BuildWithGBP
        fi

        # Create Bibliography
        biber BuildWithGBP
    done
    cd ${BasePath}
done

# Remove auxillary *.html files which are needed for epub
if ls | grep --quiet '\.html'
then
    rm *.html
fi
```

47.3. Script for extracting the scripts.

The following script can be extracted with

```
notangle -Rcreate-buildscript.sh Part6.nw > create-buildscript.sh &&t=$(date +
echo "#generated on $t" >> create-buildscript.sh
chmod ugo+x create-buildscript.sh
```

from the file *Part6.nw*.

```
393 (build-script 393)≡
cat Title.nw Part1.nw Part2.nw Part3.nw Part4.nw Part5.nw Part6.nw > BuildWithGBPg.nw
notangle -Rbuild-gbp.sh BuildWithGBPg.nw > build-gbp.sh &&t=$(date +%c)
sed --in-place \
--expression='s/This is the end, my friend[[[:cntrl:]]*/This is the end, my friend/' \
build-gbp.sh
echo "#generated on $t" >> build-gbp.sh
chmod ugo+x build-gbp.sh

%c)
notangle -Rbuild-gbp-maven-plugin.sh BuildWithGBPg.nw > build-gbp-maven-plugin.sh &&t=$(date +%c)
sed --in-place \
--expression='s/This is the end, my friend[[[:cntrl:]]*/This is the end, my friend/' \
build-gbp-maven-plugin.sh
echo "#generated on $t" >> build-gbp-maven-plugin.sh
chmod ugo+x build-gbp-maven-plugin.sh

%c)
notangle -Rbuild-gbp-webext-plugin.sh BuildWithGBPg.nw > \
build-gbp-webext-plugin.sh &&t=$(date +%c)
sed --in-place \
--expression='s/This is the end, my friend[[[:cntrl:]]*/This is the end, my friend/' \
build-gbp-webext-plugin.sh
echo "#generated on $t" >> build-gbp-webext-plugin.sh
chmod ugo+x build-gbp-webext-plugin.sh

%c)
# Remove auxillary file BuildWithGBPg.nw
rm BuildWithGBPg.nw
```

April 6, 2025

By means of the command *noweave* the German language **.nw* files are converted into **.tex* files.

```
394a  ⟨CreateTexFromNW 394a⟩≡
      function CreateTexFromNW {
        # Called by TaskSelect
        # Create *.tex files from *.nw files

        cd ${SRCDIR}

        noweave -index -delay BuildWithGBP.nw > BuildWithGBP.tex # contains preamble
        noweave -index -delay Title.nw > GBP-Title.tex
        noweave -index -delay Part1.nw > GBP-Part1.tex
        noweave -index -delay Part2.nw > GBP-Part2.tex
        noweave -index -delay Part3.nw > GBP-Part3.tex
        noweave -index -delay Part4.nw > GBP-Part4.tex
        noweave -index -delay Part5.nw > GBP-Part5.tex
        noweave -index -delay Part6.nw > GBP-Part6.tex
      }
      ⟨CreateTexFromNW1 394b⟩
```

Any existing *BuildWithGBP_html.tex* file will be removed..

```
394b  ⟨CreateTexFromNW1 394b⟩≡
      if [ -f BuildWithGBP_html.tex ] then rm BuildWithGBP_html.tex fi }
      (394a)
```

47.4. *gitlab-ci.yml* für die Salsa-CI

```

395 <gitlab-ci.yml 395>≡
  variables:
    DEPS: file make texlive noweb texlive-bibtex-extra texlive-lang-german
          texlive-lang-japanese texlive-latex-extra texlive-binaries
          texlive-extra-utils biber tidy texlive-lang-english

  stages:
    - build
    - deploy

  build:
    stage: build
    image: debian:sid
    before_script:
      - apt -y update
      - apt -y install $DEPS
    script:
      - make
    artifacts:
      paths:
        - '*.pdf'
        - '*.epub'
        - 'build-gbp-python-plugin.sh'
        - 'build-gbp-maven-plugin.sh'
        - 'build-gbp-webext-plugin.sh'
        - 'build-gbp.sh'
        - 'build-gbp-java-plugin.sh'
        - 'translation/en_US/target/*.pdf'

  pages:
    stage: deploy
    image: debian:sid
    needs:
      - build
    script:
      - mkdir -p public/de
      - mkdir -p public/en_US
      - cp *.pdf *.epub build-gbp*.sh public/
      - cp *.tex public/de/
      - cp translation/en_US/target/*.pdf public/en_US/

  artifacts:
    paths:
      - public
  only:
    - master

```

Part VII.

Anhang

List of Figures

20.1.	Information about Java-Team ¹	78
20.2.	Create access token ²	79
21.1.	Workflows [35] ³	82
28.1.	Start screen	105
28.2.	Specification of the project name.	106
28.3.	Bye	107
28.4.	No project name specified.	108
29.1.	No configuration file found.	110
29.2.	Name of the source package	112
29.3.	Specify the name of the source package	113
29.4.	Specify the correct name of the source package	114
29.5.	Correct name of the package specified.	115
29.6.	Correct name of the package specified.	116
29.7.	Name of the group specified on <i>salsa.debian.org</i>	117
29.8.	Name of the group specified on <i>salsa.debian.org</i>	117
29.9.	Name of the group specified on <i>salsa.debian.org</i>	118
29.10.	Should a Java package be built?	122
29.11.	Should an extension for Mozilla be packaged?	125
29.12.	Should a Python3 pact be built?	126
29.13.	Is there a parent Git repository?	133
29.14.	Create a new package	134
29.15.	Name and email.	137
29.16.	Debian-Maintainer OK?	141
29.17.	Name of the Debian-Maintainer	141
29.18.	Email of the Debian-Maintainer	142
29.19.	Email of the Debian-Uploaders	143
29.20.	Email of the Debian-Uploaders	144
29.21.	Debian maintainer data	147
29.22.	Add remote server	149
29.23.	Choosing the Debian Release.	153
29.24.	Name and email.	154
29.25.	GPG-Key available	157
30.1.	Query - configuration file	160

¹Source:<https://salsa.debian.org/java-team/>

²Source:<https://salsa.debian.org>

³©2016 Antoine Beaupré anarcat@debian.org, CC-BY-SA 4.0

30.2.	Query - edit configuration file.	161
30.3.	Query - Further check?	165
30.4.	Selection of the Debian branch.	168
30.5.	Selected Debian branch.	169
30.6.	There is only one Git branch	170
30.7.	No branch created	171
30.8.	Task selection.	172
31.1.	Download from salsa.debian.org	176
31.2.	There are patches	178
31.3.	Fixed? Retry?	179
31.4.	No import into patch queue	180
31.5.	PQ-Import successful	181
31.6.	No download via uscan from thunderbird.net	183
31.7.	No download via uscan from mozilla.org	184
31.8.	Download - classical or with uscan	184
31.9.	Name of the upstream URL	186
31.10.	Download (or copy)?	186
31.11.	Enter link for download	187
31.12.	Download-URL right?	188
31.13.	Correct download URL	188
31.14.	Download *.asc file	189
31.15.	Path to copy	190
31.16.	Quit program	190
31.17.	Unknown suffix	192
31.18.	Right version?	193
31.19.	Which version should be built?	193
31.20.	Will correct version be built?	194
31.21.	Quit program	194
31.22.	File debian/copyright contains section Files-Excluded	196
31.23.	Exclude files	197
31.24.	Should debian/copyright be edited?	198
31.25.	Suffix for exclusion of files	200
31.26.	Custom suffix for excluding files	201
31.27.	Warning. - No suffix specified	202
31.28.	Create orig.tar.xz	204
31.29.	mk-origtargz failed	205
31.30.	Special gbp.conf	208
31.31.	path to the gbp.conf?	209
31.32.	gbp.conf not found	210
31.33.	Check gbp.conf	211
31.34.	Check gbp.conf	212
31.35.	<i>gbp.conf</i> found twice.	213
31.36.	Different configuration files	214
31.37.	Do you want to edit <i>gbp.conf</i> in the <i>debian/</i> directory?	215
31.38.	Do you want to edit <i>gbp.conf</i> in the <i>.git/</i> directory?	216

31.39.	Dubious git tag	217
31.40.	Delete Git Tags	217
31.41.	Up to date	222
31.42.	New Version available	222
32.1.	Vuilt new revision	226
32.2.	Create data for Maven?	228
32.3.	Display Debian files	229
33.1.	There is a directory debian/patches.	252
33.2.	There is no directory debian/patches.	252
33.3.	Create patches for Debian	253
33.4.	There exists a PQ branch.	255
33.5.	Hint on the requirements for further work.	256
33.6.	The patch queue branch with patches from debian/patches has been applied.[3]	259
33.7.	Should <i>gbp pq export</i> be applied?	262
34.1.	Debian-Changelog OK?	278
34.2.	Display the first line of the <i>debian/changelog</i> file.	280
34.3.	Recent version	280
34.4.	Query the identifier	282
34.5.	More options for <i>dch</i>	283
34.6.	Something is going wrong!	286
34.7.	Selection of the branch	287
34.8.	Release-Branch	288
34.9.	Release branch of the distribution	289
34.10.	Distribution for PBuilder	291
34.11.	Start building the package	291
34.12.	Finish	292
34.13.	Information about updating the build environment	293
34.14.	Selecting the build system	294
34.15.	Check <i>.sbuildrc</i>	295
34.16.	Selection of the cow to create.	298
34.17.	Display of the version with revision number	300
34.18.	Display of the revision number	301
34.19.	Should the upstream tarball be uploaded too	302
34.20.	Display the options of <i>gbp buildbackage</i>	303
34.21.	Unsuccessful building attempt!	304
37.1.	Prepare upload of release	313
37.2.	Lintian: All Well?	316
37.3.	Older package available	317
37.4.	Uscan - OK?	318
37.5.	Uscan fails	319
37.6.	Determine differences	320

38.1.	No Changelog - No upload	326
38.2.	Changelog fir for uploading?	327
38.3.	Check branch	328
38.4.	Enter the name of the distribution	329
38.5.	Check distribution name	330
38.6.	Building for release	331
38.7.	Building for release	332
40.1.	Upload target	339
40.2.	Upload to FTP-Master - OK?	343
44.1.	Maven Plugin loaded	364
44.2.	Determine path to Maven chroot	365
44.3.	Specify path to Maven chroot	366
44.4.	Determine working directory in the Maven chroot	367
44.5.	Specify working directory in the Maven chroot	368
44.6.	Maven chroot does not exist	369

Bibliography

- [1] Creative Commons. “Attribution-ShareAlike 4.0 International”. In: *https://creativecommons.org/licenses/by-sa/4.0/legalcode*. (Oct. 15, 2013). URL: <https://creativecommons.org/licenses/by-sa/4.0/legalcode>. (Cit. on p. 3).
- [2] Free Software Foundation, Inc. “GNU General Public License 3”. In: *https://www.gnu.org/licenses/gpl-3.0.de.html*. (June 29, 2007). URL: <https://www.gnu.org/licenses/gpl-3.0.de.html>. (Cit. on p. 3).
- [3] Guido Günther. “Building Debian Packages with git-buildpackage”. In: *Git-Buildpackage* (2017). URL: <https://honk.sigxcpu.org/projects/git-buildpackage/manual-html/> (cit. on pp. 7, 22, 33, 74, 259, 261).
- [4] Norman Ramsey. “Noweb — A Simple, Extensible Tool for Literate Programming”. In: *NoWeb* (June 28, 2018). URL: <https://www.cs.tufts.edu/%5Ctextasciitilde%20nr/noweb/> (cit. on p. 10).
- [5] *Simple Packaging Tutorial*. Oct. 26, 2019. URL: <https://wiki.debian.org/SimplePackagingTutorial> (cit. on pp. 21, 23).
- [6] Debian Project. “Die Debian-Richtlinien für Freie Software (DFSG)”. German. In: *Debian-Gesellschaftsvertrag* (Apr. 26, 2004). URL: https://www.debian.org/social_contract.de.html (cit. on pp. 21, 27, 30, 31).
- [7] Ian Jackson, Christian Schwarz, and David A. Morris. “Debian Policy”. English. Version 4.6.1. In: *Debian Policy Manual* (May 22, 2022). Ed. by Die Debian-Policy-Gruppe. URL: <https://www.debian.org/doc/debian-policy/>. GNU General Public License Version 2+ (cit. on pp. 21, 23, 27, 29–31, 234, 242, 249).
- [8] Christopher Yeoh, Paul ‘Rusty’ Russell, Daniel Quinlan. “Filesystem Hierarchy Standard”. English. Version 3.0. In: *Filesystem Hierarchy Standard* (Mar. 19, 2015). Ed. by The Linux Foundation LSB Workgroup. URL: <https://www.debian.org/doc/packaging-manuals/fhs/fhs-3.0.html>. BSD (cit. on p. 21).
- [9] Ian Jackson et al. “Debian-Entwicklerreferenz”. Deutsch. Version 13.3. In: *Debian Entwicklerreferenz* (Aug. 5, 2023). Ed. by Hideki Nussbaum Lucas and Yamane and Holger Levsen. URL: <https://www.debian.org/doc/manuals/developers-reference/index.de.html>. GNU General Public License Version 2+ (cit. on pp. 21, 81–84, 100).
- [10] Osamu Aoki. “Leitfaden für Debian-Betreuer. debmake-doc”. Deutsch. In: *Debmake-Doc* (Mar. 26, 2019). URL: <https://www.debian.org/doc/devel-manuals#debmake-doc>. Expat-Lizenz (cit. on p. 22).



- [11] Josip Rodin, Osamu Aoki. “Debian-Leitfaden für Neue Paketbetreuer. New Maintainer Guide”. Deutsch. Version 1.2.43. In: *Debian-Leitfaden für Neue Paketbetreuer* (Nov. 7, 2020). Ed. by Osamu Aoki. wird ersetzt durch Aoki, Leitfaden für Debian-Betreuer. URL: <https://www.debian.org/doc/manuals/maint-guide/>. GNU General Public License Version 2+ (cit. on pp. 22, 36, 60, 279).
- [12] Axel Beckert and Frank Hofmann. “Debian-Paketmanagement”. In: *DPMB* (Feb. 7, 2021), p. 400. URL: <https://book.dpmb.org/debian-paketmanagement.chunked/index.html> (cit. on pp. 22, 24, 25).
- [13] *ReproducibleBuilds*. Dec. 6, 2020. URL: <https://reproducible-builds.org/> (cit. on p. 23).
- [14] *The experimental repository*. Nov. 15, 2020. URL: <https://wiki.debian.org/DebianExperimental> (cit. on p. 23).
- [15] *The Debian GNU/Linux FAQ*. Aug. 12, 2019. URL: <https://www.debian.org/doc/manuals/debian-faq/index.de.html> (cit. on pp. 24, 35).
- [16] *How can i help*. Feb. 7, 2021. URL: <https://wiki.debian.org/how-can-i-help> (cit. on p. 25).
- [17] Steve Langasek. “DEP-5: Machine-readable debian/copyright”. In: *Machine-readable debian/copyright* (Feb. 24, 2012). URL: <https://dep-team.pages.debian.net/deps/dep5/> (cit. on pp. 27, 28, 31, 32, 198, 233).
- [18] *TeamsFTPMaster*. Mar. 13, 2020. URL: <https://wiki.debian.org/TeamsFTPMaster> (cit. on p. 27).
- [19] *CopyrightReviewTools*. Dec. 17, 2021. URL: <https://wiki.debian.org/CopyrightReviewTolls> (cit. on p. 27).
- [20] Mathew Palmer. “Debian-Mentors FAQ”. English. In: *Debian-Wiki* (2007). URL: <https://wiki.debian.org/DebianMentorsFaq>. GNU General Public License version 2 (cit. on pp. 29, 32).
- [21] Jilayne et al. Lovejoy. “Open Source License Compliance Handbook”. In: *Open Source License* (Apr. 29, 2019). URL: <https://github.com/finos/OSLC-handbook/tree/master/output> (cit. on p. 29).
- [22] *Using Quilt*. July 22, 2020. URL: <https://wiki.debian.org/UsingQuilt> (cit. on p. 33).
- [23] Andreas Grünbacher. “How to Survive With Many Patches or Introduction to Quilt”. In: *Introduction to Quilt* (Feb. 22, 2012). URL: <http://users.suse.com/~agruen/quilt.pdf> (cit. on p. 33).
- [24] Raphael Hertzog. “DEP-3: Patch Tagging Guidelines”. In: *Patch Tagging Guidelines* (Nov. 26, 2009). URL: <https://dep-team.pages.debian.net/deps/dep3/> (cit. on pp. 33, 261).
- [25] *Git-Mailinfo -Manpage*. Apr. 20, 2020. URL: <https://manpages.debian.org/unstable/git-man/git-mailinfo.1.en.html> (cit. on p. 34).
- [26] Niels Thykier et al. “Debian Policy for Java”. In: *Debian Java Policy* (July 27, 2020). URL: <https://www.debian.org/doc/packaging-manuals/java-policy/> (cit. on pp. 41, 42, 247, 316).

- [27] Markus Koschany. “Packaging Java with Javatools”. In: <https://people.debian.org/~apo/java/tutorial.html> (Aug. 2, 2018). URL: <https://people.debian.org/%5Ctextasciitilde%7B%7Dapo/java/tutorial.html> (cit. on p. 41).
- [28] Torsten Werner twerner@debian.org Niels Thykier niels@thykier.net Javier Fernández-Sanguino Peña jfs@debian.org Sylvestre Ledru sylvestre@debian.org. “Debian Java FAQ.” In: *Debian Java FAQ*. (May 22, 2014). URL: <https://www.debian.org/doc/manuals/debian-java-faq/> (cit. on p. 41).
- [29] *Help the Java Team distribute your project*. Jan. 31, 2019. URL: <https://java.debian.net/blog/2019/01/help-the-java-team-distribute-your-project.html> (cit. on p. 41).
- [30] *Introduction to the Standard Directory Layout*. Dec. 29, 2020. URL: <https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html> (cit. on p. 43).
- [31] *Devsripts*. Nov. 28, 2020. URL: <https://manpages.debian.org/unstable/devscripts/devscripts.1.en.html> (cit. on p. 57).
- [32] *debian mit debootstrap in chroot-Umgebung installieren*. Feb. 16, 2014. URL: http://www.kai-hildebrandt.de/linux/debian%5C_chroot.html (cit. on p. 70).
- [33] *Salsa*. Feb. 22, 2023. URL: <https://wiki.debian.org/Salsa> (cit. on p. 77).
- [34] *Salsa-Doc*. Aug. 26, 2023. URL: wiki.debian.org/Salsa/Doc (cit. on p. 77).
- [35] [wiki.debian.org](https://wiki.debian.org/DebianReleases). “Debian Releases”. In: <https://wiki.debian.org/DebianReleases>. Nov. 29, 2020. URL: <https://salsa.debian.org/debian/package-cycle/raw/master/package-cycle.svg> (cit. on p. 82).
- [36] *ReleaseTeam*. Mar. 22, 2021. URL: <https://wiki.debian.org/Teams/releaseTeams> (cit. on p. 99).
- [37] *GBP-Import-orig uscan*. July 19, 2022. URL: <https://manpages.debian.org/unstable/git-buildpackage/gbp-import-orig.1.en.html> (cit. on p. 224).
- [38] [wiki.debian.org](https://wiki.debian.org/UpstreamMetadata). “Upstream METadata GAttered with YAml (UMEGAYA)”. In: <https://wiki.debian.org/UpstreamMetadata?action=recall&rev=138>. Jan. 31, 2020. URL: <https://wiki.debian.org/UpstreamMetadata?action=recall&rev=138> (cit. on p. 232).
- [39] Charles Plessy and Andreas Tille. “DEP-12: Per-package machine-readable metadata about Upstream”. In: *Per-package machine-readable metadata about Upstream* (Feb. 23, 2014). URL: <https://dep-team.pages.debian.net/deps/dep12/> (cit. on p. 232).
- [40] Machine-Readable Copyright. “Machine-readable debian/copyright file”. In: *Debian Policy* (Nov. 17, 2020). URL: <https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/%5C#stand-alone-license-paragraph> (cit. on p. 234).
- [41] Debian Projekt, ed. *Manpage uscan*. Aug. 4, 2019. URL: <https://manpages.debian.org/buster/devscripts/uscan.1.en.html> (cit. on p. 238).
- [42] *GBP-PQ*. July 1, 2020. URL: <https://manpages.debian.org/unstable/git-buildpackage/gbp-pq.1.en.html> (cit. on p. 257).

April 6, 2025

- [43] Debian, ed. *Lintian User's Manual*. Jan. 23, 2023. URL: <https://lintian.debian.org/manual/index.html> (cit. on p. 314).

Index

- ~/.bashrc, 71
- Access tokens, 78, 387
- ant, 46
- apt, 36
- Backporting, 82
- Build environments, 32
- Chroot, 61
- Chroot (maven), 44
- cme, 28
- compare-version, 35
- Configuration file, 59, 103, 108, 159, 341
- Copyright-Review, 27
- cowbuilder, 61
- cowdancer, 61
- Creative Commons, 3
- DEBEMAIL, 60
- DEBFULLNAME, 60
- debhelper, 235
- Debian Developer Reference, 21
- Debian Keyring, 341
- Debian Package, 23
- Debian Policy, 21, 27, 30
- Debian Project, 23
- debian/watch, 221, 238
- debmake, 27
- dependency, 29, 41
- Developer Reference, 21
- devscripts, 36, 57
- DFSG, 27, 30
- dh_make, 57
- Directory, 130
- Directory structure of Maven, 43
- Distribution, 9
- dpkg, 36
- dquilt, 70
- dversionmangle, 240
- ebook, 10
- EPUB Document, 10
- experimental, 23
- FHS, 21
- filenamemangle, 241
- Fingerprint, 60, 341
- Free Software, 27
- FTBFS, 61
- gbp buildpackage, 61
- gbp pq, 33
- gbp.conf, 73, 74, 221
- Geany, 10
- git, 10
- git-buildpackage, 10, 22, 57
- GNU General Public License, 3
- GPG-Key, 57, 220, 223
- home directory, 109
- Java Application, 41, 42
- Java build system, 43
- Java Library, 41, 42
- Java-FAQ, 41
- Java-Policy, 41
- javahelper, 41
- License, 3, 27
- License verification, 27, 319
- Licenses, 27, 29
- Literate Programming, 10
- Literature, 22
- Logfile, 130
- Mail-Extension, 50

April 6, 2025

main, 27
Main program, 103
Maintainer-Key, 60
Manual, 22
Maven, 43
maven, 46
mh_make, 44

New Maintianer Guide, 22
noweave, 394
noweb, 10

oldoldstable, 23
oldstable, 23
Options (uscan), 239
Original author, 28
Original source, 36

Package management system, 49
Package management tool, 234
Packaging, 7
Patching, 32
pbuilder, 61
PDF Document, 10
Perl-Format, 238
pom.xml, 43
Program flow, 103, 104
Program script, 103
Project, 105
Project name, 106
project specific, 59
Proposed-Updates, 82, 83
Publishing, 10, 173

Quick Guide, 15
quilt, 33, 70

regular expressions, 238
Release-Team, 83
Reproducibility, 23
Revision Number, 35
Rules of comparison, 36

Security Patches, 32, 82
Security-Team, 82
sign, 60
Social Contract, 21
Source code, 10

stable, 23
Standard-Version, 235
Start dialog, 103
System user, 49

testing, 23
TeX-Files, 394
Thunderbird, 50
Tools, 10
Troubleshooting, 32

unstable, 23
Updating, 49
Uploading, 325
uscan, 36, 221, 238
Utilities, 39
uversionmangle, 240

Version index, 240
Version Name, 35, 199
Versioning, 35

Watch (File), 36, 238