Create Debian Linux packages

Summary: Learn the basics of creating Debian packages for distributing programs and source code. This article shows all the necessary components of a package and how to put them together to end up with a final product.

The Debian packaging system is one of the most elegant methods of installing, upgrading, and removing software available. For all you fans of other packaging systems, before you send your flames, please note that I said "one of" and not simply "the most." Other packaging systems have their charms, but in this article I'm going to focus on the Debian packaging system. Specifically, I'm going to look at creating Debian packages so you can distribute your packages in Debian format -- or simply create packages for your own use.

I'll walk through the basic components of a package, what you need to create your own packages, and finally how to pull it all together to create a package.

Understanding Debian packages

In this section, we'll take a quick look at what Debian packages are. Debian packages are much like RPM (Red Hat Package Manager) packages, but they differ a little in the details. If you've already worked with creating RPMs, you'll probably find that it's not at all difficult to add Debian packaging to your repertoire.

What is a package, exactly?

To put it very simply, a *package* is a collection of files with instructions on what to do with them. A package usually contains a program or programs, but sometimes it has only documentation, window manager themes, or other files that are easier to distribute in an installable package.

The package contains instructions on where those files should reside in the filesystem, what libraries or other programs the contents of the package are dependent on (if any), setup instructions, and basic configuration scripts. Note that many packages cannot be used or should not be used with the default settings contained in their configuration files. With packages such as Apache, you'll still need to configure your installation after the package has been set up.

Packages usually contain precompiled software, but you can also package source code. Some admins may prefer to install from source, or your application may require customization prior to compilation, so if you're distributing software that is under a free or open source license, you may wish to create a source package as well as a "binary" package.

Note: Pre-compiled software packages are usually called "binaries," although this is something of a misnomer. Yes, executables *are* in binary format, but so are JPEGs, MP3s, and many other non-executable files. It would probably make more sense to call precompiled software "executable" packages.

All binary Debian packages consist of three basic things: a text file called debian-binary, a compressed tarball called control.tar.gz, and another compressed tarball called data.tar.gz.

The debian-binary text file contains the version number for the binary package, which should be 2.0. The control.tar.gz file contains the control file; the postinst file, which contains instructions on what to do after installing the package; and the prerm file, which contains removal instructions. control.tar.gz may also contain a file with information about configuration files for the package called conffiles and a file with the MD5 checksums for the package called md5sums.

The data.tar.gz contains the actual "payload" of the package. That is, it contains a filesystem with all the relevant files for your program that, when installed, will be placed in the appropriate spots in your system's filesystem.

If you want to see what a package looks like for yourself, download a few packages from the Debian site (see the link in the Resources later in this article) and run ar -x packagename.deb. Debian packages are simply archives of the files mentioned above.

Do I need to be a Debian Developer?

You don't need to be an official Debian Developer to create and distribute Debian-compatible packages. However, you *do* need to be a Debian Developer to actually upload packages and be a package maintainer for official Debian releases. If you're interested in becoming a Debian Developer, you can start by reading the "How You Can Help?" page on the Debian Web site (see Resources for a link).

There are a number of companies and projects that create and distribute Debian packages of their software. These are not "official" Debian packages, but they're still of great convenience to Debian users who wish to be able to install software using dpkg or APT (Advanced Packaging Tool).

What you need to get started

Obviously, you're going to need a current Debian system if you're going to create and test Debian packages. You'll also need to have something that you want to package. We'll start with the assumption that you already have those two things and go from there.

There are several packages that you'll need to have installed to build packages on a Debian system. Most of those packages will likely be installed by default, since you'll need dpkg, gcc, make, and so forth on most systems anyway. You will need the build-essential package and dpkg-dev package as well, which may or may not already be installed on your system.

Debian package naming conventions

If you've used Debian for any amount of time, you might have noticed that Debian package files all follow certain naming conventions. Every Debian binary package should have a filename that follows this format: packagename_version_arch.deb, where "packagename" is the name of the package, "version" is the package version with major, minor, and revision numbers, and "arch" is the architecture for the package.

The name of the package itself (not the filename, just the name of the package) can contain lowercase letters, numbers, and the "-" and "+" characters. (It might sound odd for a package to have a "+" in the package name, but it is used for several packages like "doc++" -- a documentation system for C/C++ code.) So, "mypackage-2++" is a perfectly valid package name, while "MY_Package" or "mypackage2.0" would not be.

Every package should also have a unique name that doesn't already exist in the Debian archive. Even though you technically do not need to adhere to this for your package to install, you should be sure that your package name does not conflict with an official Debian package. For example, if you wanted to distribute a specially patched version of Apache as an alternative to the official Debian version, you shouldn't use apache_X.X.X-i386.deb, because the Debian folks already have an *apache* package. Instead, you'd want to use something like acme-apache_X.X.X-i386.deb.

Creating a Debian package

Now that we've covered the basics, what packages are, and what you need, it's time to talk about the nuts and bolts of building packages.

Control files

You'll need to create a control file for your package. Don't worry; it's not very hard and all you need to create the file is your favorite text editor and a little information. The following is a sample control file. Note that I've included all the optional fields here for demonstration purposes -- your package is unlikely to need all of these fields.

Listing 1. A sample control file:

```
Package: acme
Version: 1.0
Section: web
Priority: optional
Architecture: all
Essential: no
Depends: libwww-perl, acme-base (>= 1.2)
Pre-Depends: perl
Recommends: mozilla | netscape
Suggests: docbook
Installed-Size: 1024
Maintainer: Joe Brockmeier <jzb@dissociatedpress.net>
Conflicts: wile-e-coyote
Replaces: sam-sheepdog
Provides: acme
Description: The description can contain free-form text
             describing the function of the program, what
             kind of features it has, and so on.
.
More descriptive text.
```

Control file fields

The syntax for the control fields is very simple: the name of the field, followed by a colon, and then the body or value of the field. So, "Fieldname: value" is all you need. Some fields require a one-word value, while others allow a free-form description or set of values.

In this section we'll look at what each field in the control file means and what kind of information you need to provide. Most of these fields are optional, so your package will *not* have to include every field described here if they aren't relevant to your package. For example, a package will not require the "Source" field if source for the package is unavailable. If your package won't conflict with any other packages, then you need not include the "Conflicts" information, and so on.

The first field in any control file is the "Package" field. In this field you'll specify the name of the package. The next field, "Version" is the version number for the package. You may use whatever version numbering scheme the original author of the package uses, but it may not include a hyphen. Anything after a hyphen is considered the Debian revision number. For example, if a program's native version number is given as 1.01-1 for some reason, the version number would be considered "1.01" with a Debian revision number of "1" -- clearly, not the result you're looking for.

The "Maintainer" field should be your full name (or company name) and an e-mail address contained within angle brackets as shown above. The "Installed-Size" is pretty self-explanatory.

Dependencies

As you probably already know, one of the nicest things about Debian packages is the dependency resolution. When you use apt-get to retrieve packages, it automatically decides whether you have what you need on your system to use the requested package, and offers to resolve any conflicts you might have or retrieve any packages you're in need of. That feature depends on the package's dependencies being well thought out and properly set up in the package.

There are several optional fields that deal with dependencies. We'll cover each of them briefly here. Be sure you've sorted out exactly which packages your package may depend on before installation -- the dependency system is only as smart as the developers giving it information.

Dependencies make up five of the fields in a control file: Depends, Enhances, Pre-Depends, Recommends, and Suggests.

The Depends field should contain the name of any packages that your package absolutely needs to work. If there's a "nice to have" package related to yours, but not required for normal operation of your package, then you should use the "Recommends" or "Suggests" fields instead. You'll note in the example above that the "Recommends" field has two separate package separated by a pipe (|). This means that either package will satisfy the dependency; both are not required.

What's the difference between "Recommends" and "Suggests," anyway? Any packages that you should probably have, but are not absolutely required, belong in the "Recommends" field. According to the policy manual, "list packages that would be found together with this one in all but unusual installations." Packages that are likely to be useful but aren't in any way required should be listed in the "Suggests" field.

For example, the DocBook XML DTD package (docbook-xml) lists the DocBook SGML (docbook) package as a recommended package, and the DocBook documentation (docbook-doc) as a suggested package. In most circumstances, if you have the XML DTD, you'll probably have the SGML DTD too -- but not in every instance. It's usually (read: always) a good idea to install documentation about a package or related programs, like the DocBook docs, but it's not necessary for normal operation of your program.

The "Enhances" field is basically stating that your package may make another package more useful. So, the docbook-doc package might have an "Enhances" field with docbook-xml as one of the packages enhanced.

The "Pre-Depends" field is reserved for special cases. When a package is listed as a "Pre-Depends," it forces the system to make sure that the named packages are completely installed before attempting to install your package.

Finally, when working with version numbers, there are several symbols you'll need to understand. You'll note that one of the packages listed as a dependency has a version number given, acme-base (>= 1.2). This means, "this package requires the package acme-base version equal to or higher than 1.2."

The relations available are strictly earlier (<<), earlier or equal (<=), equal to *only* (=), equal to or later than (>=), and strictly later than (>>).

Specifying conflicts

The "Conflicts" and "Replaces" fields specify packages that are not dependencies, but rather packages that don't belong on the system at the same time as your package.

Packages that conflict with your package outright should be listed in the "Conflicts" fields. For example, if you're packaging a modified version of Apache, you won't want the normal version of Apache to be on the system as well as yours -- so "apache" would be listed as a conflict.

Finally, the control file should be located under a directory named DEBIAN, as should the scripts we discuss in the next section. This directory won't turn up in the final Deb archive.

Scripts

Your package will also need a few scripts to be run by the system when installing and removing the package. (Remember, no matter how wonderful your program is, someone may wish to uninstall it later on down the road.)

The postinst script should contain any necessary steps to be done after installing your program. The prerm script should contain instructions on what should be done to remove the program. Both of these scripts are required (see Resources for more on this).

The final step

After all the prep work that goes into making a Debian package, the actual package creation is somewhat anticlimactic.

You'll use the dpkg command with the -b or --build option (-b and --build are the same). The syntax for creating a package is dpkg -b *directory packagename.deb* where *directory* contains the filesystem tree with all the requisite files for your program. Note, you can build the package without specifying the name of the new package, but then it will simply place the package file under directory as ".deb" -- which might lead you to believe that the package wasn't created at all.

So, for example, if your program has a configuration file that belongs in etc/, the program itself, which will live in usr/bin/, and some documentation that lives under usr/share/doc/package/, you'll recreate that filesystem tree in a directory that will be used to create your package.

Beyond the basics

That's pretty much all there is to creating a Debian package. It might seem a bit complicated, but really it's just a number of simple steps. None are too complicated; you simply have to get them all right.

There are a few intricacies of creating packages that are beyond the scope of this article. For example, if you're packaging a mail user agent or mail transport agent, there are some specific considerations that apply. This article only covers the bare basics of creating a Debian package. Make sure you check the Debian Policy Manual (see Resources for a link) to make sure you're in full compliance with the guidelines for packages.

There are several categories of programs that require extra attention, including any Web servers, daemons, Perl programs or modules, X programs, and so forth. You'll find what you need to know in the Customized Programs section of the policy manual.

Debian Binary Package Building HOWTO

Chr. Clemens Lee - <clemens@kclee.de>

This mini-HOWTO shows how to build a minimal Debian .deb package.

1. Introduction

The intended use of such a newly created archive is to install it only on your own box, not to get them into the official Debian distribution. To follow the 'official' process, please study the Debian New Maintainers' Guide.

Normal Debian packages get a proper source package, including a debian/rules file which automates the steps involved in creating the binary package. Here we just show how to package a simple shell script or binary executable into a small binary package.

BTW, I assume you know how to use 'tar', 'man', and what a '.tar.gz' file and Debian is (and how to use an editor ;-), but I assume you have never touched programs like 'ar' or 'dpkg'.

1.1. Resources on the Web

The Debian Reference gives an excellent overview as well as detailed information for everything Debian specific.

The official document for creating your own Debian packages is the Debian New Maintainers' Guide.

2. Getting Started

From the Debian Reference 2.2.2 2002-11-30: "The internals of this Debian binary package format are described in the deb(5) manual page. Because this internal format is subject to change (between major releases of Debian), always use dpkg-deb(8) for manipulating .deb files."

From the dpkg-deb man page: "dpkg-deb packs, unpacks and provides information about Debian archives. .deb files can also be manipulated with ar and tar alone if necessary. Use dpkg to install and remove packages from your system."

You might find lots of example .deb files in directory '/var/cache/apt/archives/'. With 'dpkg-deb -I somepackage.deb' you might get a general overview of what this package offers in particular. 'dpkg-deb -c somepackage.deb' lists all files which will be installed.

List content of the .deb file with 'ar tv somepackage.deb'. Use the 'x' option to extract the files.

3. Package Structure

Let's examine one example package a little bit closer. E.g. file 'parted_1.4.24-4_i386.deb' contains these three files:

$ ar tv parted_1.4.24-4_i386.deb

rw-r--r-- 0/0 4 Mar 28 13:46 2002 debian-binary

rw-r--r-- 0/0 1386 Mar 28 13:46 2002 control.tar.gz

rw-r--r-- 0/0 39772 Mar 28 13:46 2002 data.tar.gz

Now we can start to extract all files including the content of the tar files.

3.1. debian-binary

The content of this file is "2.0\n". This states the version of the deb file format. For 2.0 all other lines get ignored.

3.2. data.tar.gz

The 'data.tar.gz' file contains all the files that will be installed with their destination paths:

drwxr-xr-x root/root 0 2002-03-28 13:44:57 ./

drwxr-xr-x root/root 0 2002-03-28 13:44:49 ./sbin/

-rwxr-xr-x root/root 31656 2002-03-28 13:44:49 ./sbin/parted
drwxr-xr-x root/root 0 2002-03-28 13:44:38 ./usr/
drwxr-xr-x root/root 0 2002-03-28 13:44:41 ./usr/share/
drwxr-xr-x root/root 0 2002-03-28 13:44:38 ./usr/share/man/
drwxr-xr-x root/root 0 2002-03-28 13:44:52 ./usr/share/man/man8/
-rw-r--r-- root/root 1608 2002-03-28 13:44:37 ./usr/share/man/man8/parted.8.gz
drwxr-xr-x root/root 0 2002-03-28 13:44:41 ./usr/share/doc/
drwxr-xr-x root/root 0 2002-03-28 13:44:52 ./usr/share/doc/parted/
-rw-r--r-- root/root 1880 2002-03-07 14:20:08 ./usr/share/doc/parted/README.Debian
-rw-r--r-- root/root 1347 2002-02-27 01:40:50 ./usr/share/doc/parted/copyright
-rw-r--r-- root/root 6444 2002-03-28 13:37:33 ./usr/share/doc/parted/changelog.Debian.gz
-rw-r--r-- root/root 15523 2002-03-28 02:36:43 ./usr/share/doc/parted/changelog.gz

It must be the last file in the deb archive.

3.3. control.tar.gz

In our example this file has the following content:

-rw-r--r-- 1 root root 1336 Mar 28 2002 control
-rw-r--r-- 1 root root 388 Mar 28 2002 md5sums
-rwxr-xr-x 1 root root 253 Mar 28 2002 postinst
-rwxr-xr-x 1 root root 194 Mar 28 2002 prerm

'md5sums' contains for each file in data.tar.gz the md5sum.

In our example the content looks like this:

1d15dcfb6bb23751f76a2b7b844d3c57 sbin/parted
4eb9cc2e192f1b997cf13ff0b921af74 usr/share/man/man8/parted.8.gz
2f356768104a09092e26a6abb012c95e usr/share/doc/parted/README.Debian
a6259bd193f8f150c171c88df2158e3e usr/share/doc/parted/copyright
7f8078127a689d647586420184fc3953 usr/share/doc/parted/changelog.Debian.gz
98f217a3bf8a7407d66fd6ac8c5589b7 usr/share/doc/parted/changelog.gz

Don't worry, the 'md5sum' file as well as the 'postinst' and 'prerm' files are not mandatory for your first package. But please take a note of their existence, every proper official Debian package has them for good reasons.

'prerm' and 'postinst' seem to take care of removing old documentation files and adding a link from doc to share/doc.

```
$ cat postinst
#!/bin/sh
set -e
# Automatically added by dh_installdocs
if [ "$1" = "configure" ]; then
  if [ -d /usr/doc -a ! -e /usr/doc/parted -a -d /usr/share/doc/parted ]; then
    ln -sf ../share/doc/parted /usr/doc/parted
  fi
fi
# End automatically added section
```

```
$ cat prerm
#!/bin/sh
set -e
# Automatically added by dh_installdocs
if [ \( "$1" = "upgrade" -o "$1" = "remove" \) -a -L /usr/doc/parted ]; then
  rm -f /usr/doc/parted
fi
# End automatically added section
```

And finally the most interesting file:

```
$ cat control
Package: parted
Version: 1.4.24-4
Section: admin
Priority: optional
Architecture: i386
Depends: e2fsprogs (>= 1.27-2), libc6 (>= 2.2.4-4), libncurses5 (>= \
5.2.20020112a-1), libparted1.4 (>= 1.4.13+14pre1), libreadline4 (>= \
4.2a-4), libuuid1
Suggests: parted-doc
Conflicts: fsresize
Replaces: fsresize
Installed-Size: 76
Maintainer: Timshel Knoll <timshel@debian.org>
Description: The GNU Parted disk partition resizing program
 GNU Parted is a program that allows you to create, destroy,
 resize, move and copy hard disk partitions. This is useful
 for creating space for new operating systems, reorganizing
 disk usage, and copying data to new hard disks.
 This package contains the Parted binary and manual page.
 Parted currently supports DOS, Mac, Sun, BSD, GPT and PC98
 disklabels/partition tables, as well as a 'loop' (raw disk)
 type which allows use on RAID/LVM. Filesystems supported are
 ext2, ext3, FAT (FAT16 and FAT32) and linux-swap. Parted can
 also detect HFS (Mac OS), JFS, NTFS, ReiserFS, UFS and XFS
 filesystems, but cannot create/remove/resize/check these
 filesystems yet.
 The nature of this software means that any bugs could cause
 massive data loss. While there are no known bugs at the moment,
 they could exist, so please back up all important files before
 running it, and do so at your own risk.
Further information about the control file can be obtained via 'man 5 deb-control'.
```

4. **Hands On**

Now it is time to get practical ourselves. I have a simple shell script named 'linuxstatus' which I want to install as '/usr/bin/linuxstatus'. So first let's create a directory named 'debian' next to the file 'linuxstatus'.

```
$ mkdir -p ./debian/usr/bin
```

```
$ cp linuxstatus ./debian/usr/bin
```

4.1. **control**

Let's start with the control file. The version number must have a dash with an additional Debian package version number, e.g. '1.1-1'. If your program consists e.g. only of portable shell scripts, use 'all' as its 'Architecture'.

For 'Depends' you might need to find out to which package a certain file or program your new package relies onto belongs to. You can use 'dpkg -S <file>' for this to find this out, e.g.:

$ dkpg -S /bin/cat

coreutils: /bin/cat

Then to find out more about package 'coreutils' you can use the command 'apt-cache showpkg coreutils', which will tell you among other things the current version number that is installed on the system.

As a side note, there are two more ways to find the same information. There is a web page where you can search for Debian files: http://www.debian.org/distrib/packages. Go to the bottom of that page to fill out the web form.

Last not least there is a nice GUI application named 'kpackage', which provides convenient package browsing options and also allows to search after packages given individual file names.

'Suggests', 'Conflicts', and 'Replaces' etc. can be left out if not needed.

So here is the result of our first 'control' file:

```
Package: linuxstatus
Version: 1.1-1
Section: base
Priority: optional
Architecture: all
Depends: bash (>= 2.05a-11), textutils (>= 2.0-12), awk, procps (>= \
1:2.0.7-8), sed (>= 3.02-8), grep (>= 2.4.2-3), coreutils (>= 5.0-5)
Maintainer: Chr. Clemens Lee <clemens@kclee.de>
Description: Linux system information
 This script provides a broad overview of different  system aspects.
```

The 'control' file gets copied into a directory called 'DEBIAN' inside the other 'debian' directory.

```
$ mkdir -p debian/DEBIAN
```

```
$ find ./debian -type d | xargs chmod 755   # this is necessary on Debian Woody, don't ask me why
```

```
$ cp control debian/DEBIAN
```

If you expect your package to have a bigger audience in the future it might help to read this Writing Debian package descriptions article.

4.2. **dpkg-deb**

Now it is almost done. Just type:

```
$ dpkg-deb --build debian
dpkg-deb: building package `linuxstatus' in `debian.deb'.
$ mv debian.deb linuxstatus_1.1-1_all.deb
```

Uh, that was all easier than expected. Now we just have to install this package on our box and we are done:

```
root# dpkg -i ./linuxstatus_1.1-1_all.deb
```

Type 'linuxstatus' or 'ls -l /usr/bin/linuxstatus' to see if it worked. If you don't like your package any more, just type 'dpkg -r linuxstatus' and check again that the package is deinstalled. If you install a newer version you don't have to remove the old one first, thought.

If you are curious about the version numbering scheme and naming conventions for a Debian package, have a read at this section in The Debian Reference.

5. **Double Check**

Now that you have gotten a first impression and build your own binary package, it is time to get a little bit more serious and have a look at the quality of the package that we have produced.

5.1. **lintian**

Luckily for us the Debian project provides a 'lint' like tool for checking Debian packages. This tool is named 'lintian'. If you have not installed it yet on your system, this is a good moment (apt-get install lintian).

Now we use this little treasure tool on our new package file:

```
$ lintian linuxstatus_1.1-1_all.deb
E: linuxstatus: binary-without-manpage linuxstatus
E: linuxstatus: no-copyright-file
W: linuxstatus: prerm-does-not-remove-usr-doc-link
W: linuxstatus: postinst-does-not-set-usr-doc-link
```

Uh, doesn't look so perfect. We miss a man page, copyright file, and also those 'prerm' and 'postinst' scripts.

5.2. **Minimal Documentation**

This is not the place to say much about writing and creating man pages, there are many books that have one or another chapter related to this topic and there is also The Linux MAN-PAGE-HOWTO online. So lets do a little time warp and assume you have now a perfect man page for your script at location ./man/man1/linuxstatus.1.

The same for a 'copyright' file. You can find enough examples under the /usr/share/doc directory with this command: find /usr/share/doc -name "copyright"

So here is our own example of a '`copyright`' file:

```
linuxstatus
Copyright: Chr. Clemens Lee <clemens@kclee.de>
2002-12-07
```

```
The home page of linuxstatus is at:
http://www.kclee.de/clemens/unix/index.html#linuxstatus

The entire code base may be distributed under the terms of the GNU General
Public License (GPL), which appears immediately below.  Alternatively, all
of the source code as any code derived from that code may instead be
distributed under the GNU Lesser General Public License (LGPL), at the
choice of the distributor. The complete text of the LGPL appears at the
bottom of this file.

See /usr/share/common-licenses/(GPL|LGPL)
```

For the 'prerm' and 'postinst' scripts we copy one to one the examples from the 'parted' package above into files with the same name in our own project directory. These files should work for us just as well.

Now we create the debian package again. In the 'control' file we first increase the version number from 1.1-1 to 1.2-1 (since we have written a new man page we increase our internal release number). We also need to copy the new files to their appropriate places:

```
$ mkdir -p ./debian/usr/share/man/man1
$ mkdir -p ./debian/usr/share/doc/linuxstatus
$ find ./debian -type d | xargs chmod 755
$ cp ./man/man1/linuxstatus.1 ./debian/usr/share/man/man1
$ cp ./copyright ./debian/usr/share/doc/linuxstatus
$ cp ./prerm ./postinst ./debian/DEBIAN
$ gzip --best ./debian/usr/share/man/man1/linuxstatus.1
$ dpkg-deb --build debian
dpkg-deb: building package `linuxstatus' in `debian.deb'.
$ mv debian.deb linuxstatus_1.2-1_all.deb
```

Gzip is necessary because lintian expects man page files to be compressed as small as possible.

5.3. **fakeroot**

Now lets see if our package has become a better Debian citizen:

```
$ lintian linuxstatus_1.2-1_all.deb
E: linuxstatus: control-file-has-bad-owner prerm clemens/clemens != root/root
E: linuxstatus: control-file-has-bad-owner postinst clemens/clemens != root/root
E: linuxstatus: bad-owner-for-doc-file usr/share/doc/linuxstatus/ clemens/clemens !=
root/root
E: linuxstatus: bad-owner-for-doc-file usr/share/doc/linuxstatus/copyright
clemens/clemens != root/root
E: linuxstatus: debian-changelog-file-missing
```

Ups, new complains. OK, we will not give up. Actually most errors seem to be the same problem. Our files are all packaged for user and group 'clemens', while I assume most people would prefer having them installed as 'root/root'. But this is easily fixed using the tool 'fakeroot'. So lets fix and check this quickly (while ignoring the changelog issue):

```
$ fakeroot dpkg-deb --build debian
dpkg-deb: building package `linuxstatus' in `debian.deb'.
$ mv debian.deb linuxstatus_1.2-1_all.deb
$ lintian linuxstatus_1.2-1_all.deb
E: linuxstatus: debian-changelog-file-missing
```

Fine, but we have yet another file to add to the package.

5.4. **More Documentation**

Let me tell you already that next to a 'changelog' file in the 'doc/linuxstatus' directory a 'changelog.Debian' file is also required. Both should be gzipped as well.

Here are two example files, 'changelog':

```
linuxstatus (1.2-1)

  * Made Debian package lintian clean.

 -- Chr. Clemens Lee <clemens@kclee.de>  2002-12-13
```

and 'changelog.Debian':

```
linuxstatus Debian maintainer and upstream author are identical.
Therefore see also normal changelog file for Debian changes.
```

The Debian Policy file has more details regarding the format of the changelog file.

Now hopefully our last step will be:

```
$ cp ./changelog ./changelog.Debian ./debian/usr/share/doc/linuxstatus
$ gzip --best ./debian/usr/share/doc/linuxstatus/changelog
$ gzip --best ./debian/usr/share/doc/linuxstatus/changelog.Debian
$ fakeroot dpkg-deb --build ./debian
dpkg-deb: building package `linuxstatus' in `debian.deb'.
$ mv debian.deb linuxstatus_1.2-1_all.deb
$ lintian linuxstatus_1.2-1_all.deb
```

Ah, we get no more complains :-). As root you can install now this package over the old one, again with the standard 'dpkg -i' command.

```
root# dpkg -i ./linuxstatus_1.2-1_all.deb
(Reading database ... 97124 files and directories currently installed.)
Preparing to replace linuxstatus 1.1-1 (using linuxstatus_1.2-1_all.deb) ...
Unpacking replacement linuxstatus ...
Setting up linuxstatus (1.2-1) ...
```

6. **Summary**

Not to get confused, let us recapture all steps we have taken to build our binary Debian package.

Prerequisite files:

1.one or more binary executable or shell script files

2.a man page for each executable file

3.a 'control' file

4.a 'copyright' file

5.a 'changelog' and 'changelog.Debian' file

Setup temporary 'debian' directories:

1.create 'debian/usr/bin' directory (or wherever you plan to place your executable files)

2.create 'debian/usr/share/man/man1' (or whatever section your man page belongs into)

3.create 'debian/DEBIAN' directory

4.create 'debian/usr/share/doc/<package_name>'

5.make sure all sub directories of 'debian' have file permission 0755

Copy files into temporary 'debian' tree:

1.copy executable file into 'debian/usr/bin' directory (or wherever you plan to place your executable files)

2.copy man page file into 'debian/usr/share/man/man1' directory

3.copy 'control' file into 'debian/DEBIAN' directory

4.copy 'copyright', 'changelog', and 'changelog.Debian' files into 'debian/usr/share/doc/<package_name>'

5.gzip man page, 'copyright', 'changelog', and 'changelog.Debian' files with option '--best' inside the temporary 'debian' tree

Build and check binary Debian package:

1.invoke 'dpkg-deb --build' using 'fakeroot' on the 'debian' directory

2.rename resulting 'debian.deb' file to its final package name including version and architecture information

3.check resulting .deb package file for Debian policy compliance using 'lintian'

7. **What Else**

There are many details which have not been covered here, like how to distribute Unix daemons, configuration files and much more.

But most important, I want to emphasize again that for Debian maintainers, packages are source packages, not binary packages. They never interact directly with the internals binary packages. In fact only 'dpkg-deb' and 'dpkg' developers need to know what they are. In fact it is not recommended to do so.

If a developer were to explain someone how to build a Debian package, he will certainly explain how to make a source package and how to build it.

On the other hand, not every developer wants to submit his software to Debian (yet), but still wants to profit from the advantages a packaging system like 'dpkg' offers without releasing package source code. Personally I will release my freeware projects still as tar.gz files with source code etc. for all kind of platforms, while I plan to offer more and more '.deb' packages for the convenience of Debian users who just want to install and use my software.

If someone wants to do the next step to submit a software package to Debian, you have to move on to study the Debian New Maintainers' Guide as well as the Debian Policy Manual first. On your undertaking to create a Debian source package, also have a look at the debian-mentors mailing list to see experienced and beginning Debian developers interacting with each other and tackling similar problem you might encounter.