Debian Packaging

Matthew Palmer

mpalmer@debian.org

Overview

Overview

- Structure of Binary Packages

Overview

- Structure of Binary Packages
- Structure of Source Packages

Overview

- Structure of Binary Packages
- Structure of Source Packages
- Turning one into the other

Overview

- Structure of Binary Packages
- Structure of Source Packages
- Turning one into the other
- Interesting Problems in Packaging

Overview

- Structure of Binary Packages
- Structure of Source Packages
- Turning one into the other
- Interesting Problems in Packaging
  - Multiple Binary Packages

Overview

- Structure of Binary Packages
- Structure of Source Packages
- Turning one into the other
- Interesting Problems in Packaging
  - Multiple Binary Packages
  - Library Packages

Overview

- Structure of Binary Packages
- Structure of Source Packages
- Turning one into the other
- Interesting Problems in Packaging
  - Multiple Binary Packages
  - Library Packages
  - Patch Management

Overview

- Structure of Binary Packages
- Structure of Source Packages
- Turning one into the other
- Interesting Problems in Packaging
  - Multiple Binary Packages
  - Library Packages
  - Patch Management
  - The sid dilemma

Binary Packages

Every `.deb` is actually an `ar` archive, containing `data.tar.gz` (files for the filesystem), `control.tar.gz` (maintainer scripts and other metadata), and `debian-binary` (containing the packaging version, currently 2.0).
You can manually create or manipulate Debian packages using standard Unix tools – one of the advantages of the format. Far better, though, is to use the tools designed for the purpose...

- `dpkg --info x.deb` – Package metadata
- `dpkg --contents x.deb` – File listing
- `dpkg --unpack x.deb` – Extract the package, but don't run the configure scripts
- `dpkg --install x.deb` – Extract, configure, etc
- `dpkg --help` – Lots and lots and lots of options

Package Metadata

Binary packages have several pieces of metadata associated with them, viewable with `dpkg --info`. Of particular interest:

Package Metadata

Binary packages have several pieces of metadata associated with them, viewable with `dpkg --info`. Of particular interest:

- `Package:` the actual package name. The filename of the package is totally unimportant.

Package Metadata

Binary packages have several pieces of metadata associated with them, viewable with `dpkg --info`. Of particular interest:

- `Package:` the actual package name. The filename of the package is totally unimportant.
- `Source:` the source package that the binary package was built from.

Package Metadata

Binary packages have several pieces of metadata associated with them, viewable with `dpkg --info`. Of particular interest:

- `Package:` the actual package name. The filename of the package is totally unimportant.
- `Source:` the source package that the binary package was built from.
- `Version:` Oddly enough, the version of the package. Usually comprised of upstream version (portion before the hyphen) and Debian version (portion after the hyphen).

Package Metadata

Binary packages have several pieces of metadata associated with them, viewable with `dpkg --info`. Of particular interest:

- `Package:` the actual package name. The filename of the package is totally unimportant.
- `Source:` the source package that the binary package was built from.
- `Version:` Oddly enough, the version of the package. Usually comprised of upstream version (portion before the hyphen) and Debian version (portion after the hyphen).
- `Architecture:` what CPU the package is built for.

Package Metadata

Binary packages have several pieces of metadata associated with them, viewable with `dpkg --info`. Of particular interest:

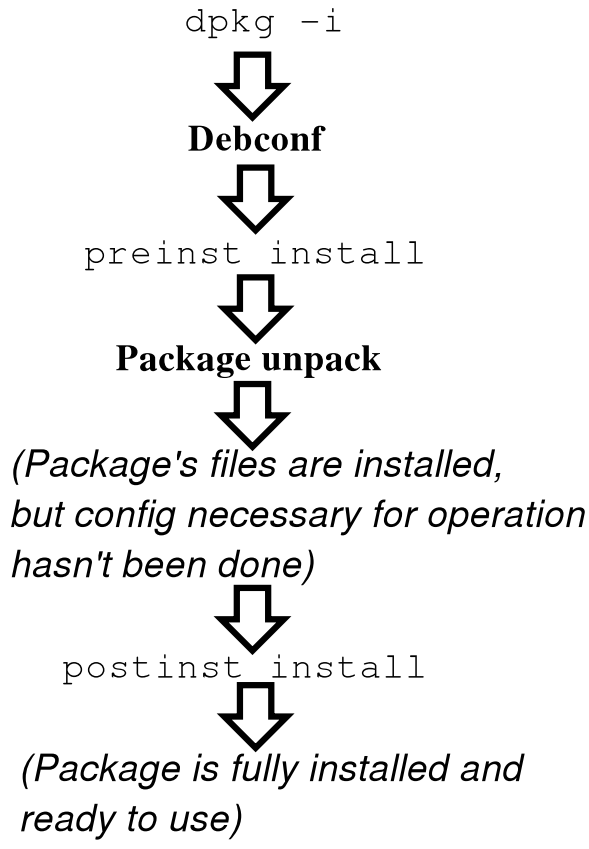
- `Package:` the actual package name. The filename of the package is totally unimportant.
- `Source:` the source package that the binary package was built from.
- `Version:` Oddly enough, the version of the package. Usually comprised of upstream version (portion before the hyphen) and Debian version (portion after the hyphen).
- `Architecture:` what CPU the package is built for.
- `Depends, Recommends, Suggests, Replaces, Conflicts, Enhances:` Fields describing various relationships with other packages.

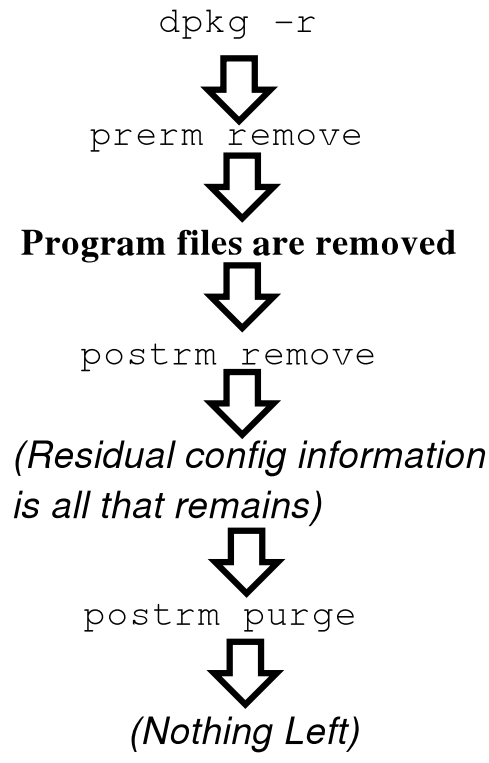
Maintainer Scripts

The package installation and removal scripts. Typically written in Bourne Shell or Perl, they perform any required configuration and deconfiguration on package installation and removal. The four standard scripts are `preinst`, `postinst`, `prerm`, and `postrm`. There is also a debconf pre-installation script, called `config`, which is supposed to ask the user all sorts of questions, whose answers are used in the other maintainer scripts.

Installation Flow

(See also Chapter 6 of Debian Policy)





Source Packages

Source Packages

- The overall control file is the `.dsc` – a description of the source package and appropriate fields to describe build parameters.

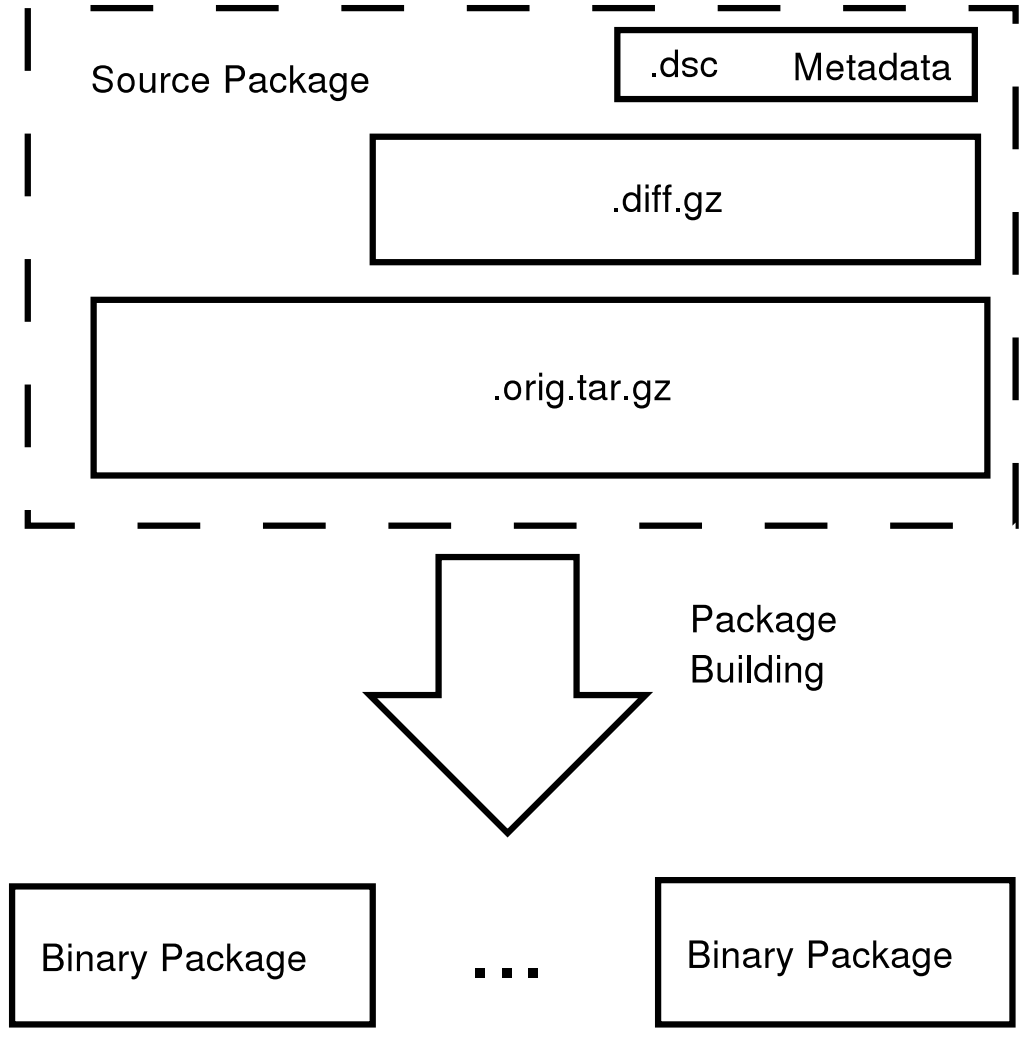
Source Packages

- The overall control file is the `.dsc` – a description of the source package and appropriate fields to describe build parameters.
- Usually an `.orig.tar.gz`, which should be the original source code as provided by upstream, or close to it; and a `.diff.gz` which contains all of the changes to the upstream source made for Debian.

Source Packages

- The overall control file is the `.dsc` – a description of the source package and appropriate fields to describe build parameters.
- Usually an `.orig.tar.gz`, which should be the original source code as provided by upstream, or close to it; and a `.diff.gz` which contains all of the changes to the upstream source made for Debian.
- “Native” packages have no orig and diff split, but simply put everything into a single tarball. Do not use this package format for most packages.

A Package Diagram



A source package unpacked

Most of the changes in a "Debianised" source package are localised in the `debian` directory. This directory contains the files which control both how the package gets built, what the binary packages are called, and the maintainer scripts used during binary package installation.

Other changes are often needed to source packages, to make them policy conformant or to fix bugs, but these should be minimised by passing changes upstream for integration.

debian/rules

It sure does!

debian/rules

It sure does!

More specifically, `debian/rules` is the file which controls how a Debian package is built. It is typically a makefile, defining several targets, corresponding to the various stages of the build process.

debian/rules

It sure does!
More specifically, `debian/rules` is the file which controls how a Debian package is built. It is typically a makefile, defining several targets, corresponding to the various stages of the build process.

- `configure` Does any necessary pre-build configuration, like running `./configure` with appropriate options. (optional)

debian/rules

It sure does!
More specifically, `debian/rules` is the file which controls how a Debian package is built. It is typically a makefile, defining several targets, corresponding to the various stages of the build process.

- `configure` Does any necessary pre-build configuration, like running `./configure` with appropriate options. (optional)
- `build` compiles the package from source. (required)

debian/rules

It sure does!
More specifically, `debian/rules` is the file which controls how a Debian package is built. It is typically a makefile, defining several targets, corresponding to the various stages of the build process.

- `configure` Does any necessary pre-build configuration, like running `./configure` with appropriate options. (optional)
- `build` compiles the package from source. (required)
- `install` copies/moves files from where they were placed when built into the installation tree. (optional)

debian/rules

It sure does!
More specifically, `debian/rules` is the file which controls how a Debian package is built. It is typically a makefile, defining several targets, corresponding to the various stages of the build process.

- `configure` Does any necessary pre-build configuration, like running `./configure` with appropriate options. (optional)
- `build` compiles the package from source. (required)
- `install` copies/moves files from where they were placed when built into the installation tree. (optional)
- `binary`, `binary-arch`, and `binary-indep` create the binary packages. `binary` typically invokes `binary-arch` and `binary-indep` indirectly.

debian/rules

It sure does!
More specifically, `debian/rules` is the file which controls how a Debian package is built. It is typically a makefile, defining several targets, corresponding to the various stages of the build process.

- `configure` Does any necessary pre-build configuration, like running `./configure` with appropriate options. (optional)
- `build` compiles the package from source. (required)
- `install` copies/moves files from where they were placed when built into the installation tree. (optional)
- `binary`, `binary-arch`, and `binary-indep` create the binary packages. `binary` typically invokes `binary-arch` and `binary-indep` indirectly.
- `clean` returns the package to it's original (pre-build) state.

debian/control

This file describes the source package and all binary packages, including their section, architecture, and build and installation dependencies and conflicts.

The .changes file

This is an important file, mainly used for the upload stage of package development. It describes the changes represented by a given version of a package – the package affected, files supplied, version, bugs closed, the maintainer, the uploader, and so on. Primarily used by the upload queue processing software, but also useful as a summary of ongoing development.

Building a package

Building a package

The minimum requirement is that the `binary` target of `debian/rules` produce the binary packages specified in `debian/control`. So, in principle, all that someone needs to do is change into the root of the package source tree and run `debian/rules binary` as root.

Building a package

The minimum requirement is that the `binary` target of `debian/rules` produce the binary packages specified in `debian/control`. So, in principle, all that someone needs to do is change into the root of the package source tree and run `debian/rules binary` as root.
How exactly the package is built is left up to the package maintainer when writing the rules file. So, you can have a lot of shell script fragments doing sick and twisted things to produce a package.

Building a package

The minimum requirement is that the `binary` target of `debian/rules` produce the binary packages specified in `debian/control`. So, in principle, all that someone needs to do is change into the root of the package source tree and run `debian/rules binary` as root.
How exactly the package is built is left up to the package maintainer when writing the rules file. So, you can have a lot of shell script fragments doing sick and twisted things to produce a package. There are, however, several much better ways of writing `debian/rules` files.

Building a package

The minimum requirement is that the `binary` target of `debian/rules` produce the binary packages specified in `debian/control`. So, in principle, all that someone needs to do is change into the root of the package source tree and run `debian/rules binary` as root.

How exactly the package is built is left up to the package maintainer when writing the rules file. So, you can have a lot of shell script fragments doing sick and twisted things to produce a package. There are, however, several much better ways of writing `debian/rules` files.

A security note: the `fakeroot` program (from the `fakeroot` package) can be used to simulate operations requiring root privileges.

Helpers and their Usefulness

The most commonly used build help system is `debhelper`, in the package of the same name and documented in `debhelper`(7) and the great many `dh_*` commands available.

Each `dh_` script performs a particular small, well defined action, such as “install files related to init scripts” and “create defined symlinks”. You can assemble a list of appropriate `dh_` scripts to suit your particular package.

Config files for debhelper scripts are kept in the `debian/` directory, and their use is described in the relevant `dh_` manpage.

Automatic Packaging

`dh_make` (package: `dh-make`) is often used as an initial packaging tool. It creates a `debian/` directory full of useful examples and almost-ready-to-go scripts. It is an excellent starting point for most debianisation efforts.

`dh-make-perl` and `dh-make-php` are `dh_make` alternatives tuned for the particular requirements of packaging Perl modules and PEAR/PECL modules, respectively.

More Advanced Helpers

In the spirit of Free Software, there are several advanced build helpers:

More Advanced Helpers

In the spirit of Free Software, there are several advanced build helpers:

- `dpatch`;

More Advanced Helpers

In the spirit of Free Software, there are several advanced build helpers:

- `dpatch`;
- `dbs`;

More Advanced Helpers

In the spirit of Free Software, there are several advanced build helpers:

- `dpatch`;
- `dbs`;
- `cdbs`.

More Advanced Helpers

In the spirit of Free Software, there are several advanced build helpers:

- `dpatch`;
- `dbs`;
- `cdbs`.

And probably several others I've forgotten or don't know about.

Build-time Helpers

When it comes time to run the build scripts, there are several very useful scripts you can use to automate part or all of the larger build process – the `debian/rules binary` call, and associated scaffolding.

Build-time Helpers

When it comes time to run the build scripts, there are several very useful scripts you can use to automate part or all of the larger build process – the `debian/rules binary` call, and associated scaffolding.

- `dpkg-buildpackage` – Cleans the source, builds the binary packages, build description (`.dsc`) and `.changes` files, and (by default) signs the package ready for upload.

Build-time Helpers

When it comes time to run the build scripts, there are several very useful scripts you can use to automate part or all of the larger build process – the `debian/rules binary` call, and associated scaffolding.

- `dpkg-buildpackage` – Cleans the source, builds the binary packages, build description (`.dsc`) and `.changes` files, and (by default) signs the package ready for upload.
- `debuild` – Wraps `dpkg-buildpackage`, adding extra useful bits like automatic lintian/linda checks.

Build-time Helpers

When it comes time to run the build scripts, there are several very useful scripts you can use to automate part or all of the larger build process – the `debian/rules binary` call, and associated scaffolding.

- `dpkg-buildpackage` – Cleans the source, builds the binary packages, build description (`.dsc`) and `.changes` files, and (by default) signs the package ready for upload.
- `debuild` – Wraps `dpkg-buildpackage`, adding extra useful bits like automatic lintian/linda checks.
- `pbuilder` – Builds a package in a clean chroot environment.

Build-time Helpers

When it comes time to run the build scripts, there are several very useful scripts you can use to automate part or all of the larger build process – the `debian/rules binary` call, and associated scaffolding.

- `dpkg-buildpackage` – Cleans the source, builds the binary packages, build description (`.dsc`) and `.changes` files, and (by default) signs the package ready for upload.
- `debuild` – Wraps `dpkg-buildpackage`, adding extra useful bits like automatic lintian/linda checks.
- `pbuilder` – Builds a package in a clean chroot environment.
- `cvs-buildpackage` – Pulls a debian release out of CVS and automatically builds it. There are also analogous `svn-buildpackage` and `tla-buildpackage` scripts.

Multiple Binary Packages

Multiple Binary Packages

- Each binary package must have it's own stanza in the `debian/control` file.

Multiple Binary Packages

- Each binary package must have it's own stanza in the `debian/control` file.
- The `binary-arch` and `binary-indep` targets must build the relevant packages.

Multiple Binary Packages

- Each binary package must have it's own stanza in the `debian/control` file.
- The `binary-arch` and `binary-indep` targets must build the relevant packages.
- debhelper scripts should be given the -a (act on all arch-specific packages) or -i (act on all arch-independent package) options, and any debhelper control files should have the bianry package name prepended.

Multiple Binary Packages

- Each binary package must have it's own stanza in the `debian/control` file.
- The `binary-arch` and `binary-indep` targets must build the relevant packages.
- debhelper scripts should be given the -a (act on all arch-specific packages) or -i (act on all arch-independent package) options, and any debhelper control files should have the bianry package name prepended.
- Files for each binary package should be placed in `debian/[packagename]`. Two ways to do this:

Multiple Binary Packages

- Each binary package must have it's own stanza in the `debian/control` file.
- The `binary-arch` and `binary-indep` targets must build the relevant packages.
- debhelper scripts should be given the -a (act on all arch-specific packages) or -i (act on all arch-independent package) options, and any debhelper control files should have the bianry package name prepended.
- Files for each binary package should be placed in `debian/[packagename]`. Two ways to do this:
  1. Have the `install` target install into `debian/tmp` and then use either `dh_movefiles` or `dh_install` put the files into the relevant package-specific directories; or

Multiple Binary Packages

- Each binary package must have it's own stanza in the `debian/control` file.
- The `binary-arch` and `binary-indep` targets must build the relevant packages.
- debhelper scripts should be given the -a (act on all arch-specific packages) or -i (act on all arch-independent package) options, and any debhelper control files should have the bianry package name prepended.
- Files for each binary package should be placed in `debian/[packagename]`. Two ways to do this:
  1. Have the `install` target install into `debian/tmp` and then use either `dh_movefiles` or `dh_install` put the files into the relevant package-specific directories; or
  2. Copy the files in manually, either directly or via `debian/tmp`.

Library Packaging

Library Packaging

- Multiple packages, with some added spice.

Library Packaging

- Multiple packages, with some added spice.
- Primary package named `lib[name][ver]`, contains the shared object, a `shlibs` control file, and a `postinst` that invokes `ldconfig` at the configure stage.

Library Packaging

- Multiple packages, with some added spice.
- Primary package named `lib[name][ver]`, contains the shared object, a `shlibs` control file, and a `postinst` that invokes `ldconfig` at the configure stage.
- Development package, usually named `lib[name]-dev`, with a static version of the library, headers, development manpages, and a `.so` symlink.

Patch Management

Patch Management

- Modify `debian/rules` to support dpatch (see the *DPATCH IN DEBIAN PACKAGES* section of `dpatch`(7));

Patch Management

- Modify `debian/rules` to support dpatch (see the *DPATCH IN DEBIAN PACKAGES* section of `dpatch`(7));
- Place patches in `debian/patches` (it's not just a straight diff; there is a shell script fragment that gets prepended, see `/u/s/d/dpatch/examples/sample.00template.gz`);

Patch Management

- Modify `debian/rules` to support dpatch (see the *DPATCH IN DEBIAN PACKAGES* section of `dpatch`(7));
- Place patches in `debian/patches` (it's not just a straight diff; there is a shell script fragment that gets prepended, see `/u/s/d/dpatch/examples/sample.00template.gz`);
- List patches to apply in `debian/patches/00list`; and

Patch Management

- Modify `debian/rules` to support dpatch (see the *DPATCH IN DEBIAN PACKAGES* section of `dpatch`(7));
- Place patches in `debian/patches` (it's not just a straight diff; there is a shell script fragment that gets prepended, see `/u/s/d/dpatch/examples/sample.00template.gz`);
- List patches to apply in `debian/patches/00list`; and
- Tally Ho!

The sid dilemma

You don’t want to run sid on your workstation, but you need to build your packages in sid before uploading them. Or, you’re running sid on your workstation but need to backport some packages to work on your woody-based server. Oh, the humanity!

The sid dilemma

You don’t want to run sid on your workstation, but you need to build your packages in sid before uploading them. Or, you’re running sid on your workstation but need to backport some packages to work on your woody-based server. Oh, the humanity!

`pbuilder` to the rescue! Takes a source package description and runs the build process in a pre-built chroot of whatever release you wish to target.

Calling `pbuilder`

- Create the chroot: `pbuilder create --basetgz /var/chroots/woody.tgz --distribution woody`

Calling pbuilder

- Create the chroot: `pbuilder create --basetgz /var/chroots/woody.tgz --distribution woody`
- Build your package in the chroot: `pbuilder build --basetgz /var/chroots/woody.tgz mypackage_1.1-1.dsc --buildresult ‘pwd‘`

Calling `pbuilder`

- Create the chroot: `pbuilder create --basetgz /var/chroots/woody.tgz --distribution woody`
- Build your package in the chroot: `pbuilder build --basetgz /var/chroots/woody.tgz mypackage_1.1-1.dsc --buildresult ‘pwd‘`
- Periodically, you should update your chroot (especially important when building for sid): `pbuilder update --basetgz /var/chroots/woody.tgz`

Packages of Interest

Documentation: `maint-guide`, `developers-reference`, `debian-policy`, `build-essential`

debian/rules helpers: `debhelper`, `dpatch`, `cbs`, `cdbs`

Build-time helpers: `devscripts`, `dpkg-dev`, `{cvs,svn,tla}-buildpackage`, `pbuilder`, `fakeroot`

Packaging templates: `dh-make`, `dh-make-perl`, `dh-make-php`

Quality Checks: `lintian`, `linda`

Acknowledgements

- “If I have packaged farther, it is because I have hacked on the shoulders of giants.”
- Brown Brothers, for a very nice Port to sip while preparing slides.
- Meat Loaf, for the best possible hacking music.
- My employer for letting my hack on both Debian and these slides on company time.