CDBS Documentation

COLLABORATORS			
	TITLE : CDBS Documentation		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Marc (Duck) Dequènes and Arnaud (Rtp) Patard	2005-03-29	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.1.0	2005-04-03	First Public Release (for CDBS V0.4.27-3)	
0.1.1	2005-06-07	Updated for CDBS V0.4.30 (perl class build dependency management, **cdbs-edit-patch** script)	
0.1.2	2005-07-05	Added DEB_CONFIGURE_SCRIPT_ENV usage warning, fixed typo.	
0.1.3	2005-09-16	Added info about dpatch extension requirements (additional include + include order). Added warning and workaround for `DEB_AUTO_UPDATE_DEBIAN_CONTROL` problems (see #311724). Fixed typos.	
0.1.4	2005-10-02	Added info about quilt extension for patching sources.	
0.2.0	2006-01-05	Added info about Ruby classes (Team & **setup.rb**). Reordered makefile and autotools class, and explained relationship. Document extraordinary use of `DEB_MAKE_ENVVARS` in autotools class.	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.3.0	2006-04-23	Fixed typo (s/foo-date/foo-data/ reported by tioui). Warned of possible breakage if spaces in `CURDIR` (see #306941). Removed hacks in examples because of #300149, #284231 and #239128 / #341275. Updated for CDBS 0.4.37 (document `DEB_MAKE_MAKEFILE`, and special case when `DEB_MAKE_CLEAN_TARGET` can be empty + **dh_installmime** and **dh_installcatalogs** now called in the debhelper class + s/`DEB_ANT_TEST_TARGET`/`DEB_ANT_CHECK_TARGET`/ which was a mistake in code, see #307813 + document `DEB_CLEAN_EXCLUDE` + default compat mode changed to 5 and `DEB_DH_STRIP_ARGS` usage too). Warn rules MUST come after CDBS includes (see #273835). Improved documentation of common build options.	
0.4.0	2006-04-24	Updated for CDBS 0.4.39 (ability to use uuencoded patches + **dh_installudev** now called in debhelper class + KDE class improvements + `config.*` left over autotools files not removed anymore + new `DEB_DH_COMPAT_DISABLE` variable + improved scrollkeeper and Python cleanup + updated common variables available in `debian/rules`). Updated some examples accordingly. Improved part on compat. Improved fixes related to `DEB_AUTO_UPDATE_DEBIAN_CONTROL` problems.	
0.5.0	2009-04-18	Updated for CDBS 0.4.56 (Python class update + updated `DEB_CONFIGURE_SCRIPT_ENV` usage + compat level 7 + documented new cmake and qmake classes). Made clear **cdbs-edit-patch** is for the simple-patchsys patch system. Added **dh_icons** to list of debhelper commands handled by the GNOME class. Improved documentation of `DEB_AUTO_UPDATE_*` variables. Improved docbook tags a lot. Fixed punctuation a bit.	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.6.0	2010-06-06	Updated for CDBS 0.4.84 (updated supported tarball extensions + deprecate kde class + `DEB_ACLOCAL_ARGS` + parallel builds + Python `.egg-info` cleanup + `DEB_PYTHON_PRIVATE_MODULES_DIRS_xxx` + utils rules + `DEB_MAINTAINER_MODE` + upstream-tarball rule + python-sugar class + build flavors for makefile and autotools classes + `CDBS_BUILD_DEPENDS_DELIMITER` + python-autotools class + scons class + perl-makemaker and perl-build classes + cross-building notes). Clarify `DEB_AUTO_CLEANUP_RCS` removal and replacement. Reorganized maintainer tools in a new chapter. Major rework for variables, rules, and debhelper related documentation. Updated examples. Removed a few deprecated things. Fixed several typos and improved docbook tags usage again.	
0.6.1	2011-03-24	Fixed missing "::" at the end of the patsubst lines in the "Class for Python distutils" chapter.	

Contents

1 Introduction **1**

1.1 A bit of history . . . 1

1.2 Why CDBS ? . . . 1

2 First steps **2**

2.1 Convert a package to CDBS . . . 2

2.2 Basic settings and available variables . . . 2

2.3 Custom build rules . . . 4

2.3.1 Examples . . . 4

2.4 Common Build Options . . . 6

3 Common Tasks **7**

3.1 Automatic tarball management . . . 7

3.2 Patching sources . . . 7

3.2.1 Rules for simple-patchsys . . . 7

3.2.2 Rules for dpatch . . . 8

3.2.3 Rules for quilt . . . 8

3.3 Using Debhelper . . . 9

3.3.1 Not managing Debhelper . . . 9

3.3.2 Debhelper customization examples . . . 11

3.4 Cross-Building . . . 12

4 Specific Tasks **13**

4.1 Packaging applications with a generic build-system . . . 13

4.1.1 Class for Makefile . . . 13

4.1.1.1 Classic Build . . . 13

4.1.1.2 Multiple "flavors" Build . . . 14

4.1.2 Class for Autotools . . . 14

4.1.2.1 Classic Build . . . 14

4.1.2.2 Multiple "flavors" Build . . . 16

4.1.3 Class for CMake . . . 16

4.1.4 Class for qmake . . . 16
4.1.5 Class for SCons . . . 16
4.2 Packaging Perl applications . . . 17
4.2.1 Subclass for Makefile (ExtUtils::MakeMaker) . . . 17
4.2.2 Class for Module::Build . . . 17
4.2.3 Customizations common to all Perl classes . . . 18
4.3 Packaging Python applications . . . 18
4.3.1 Class for Python distutils . . . 19
4.3.2 Subclass for autotools . . . 19
4.3.3 Class for Python Sugar . . . 20
4.3.4 Customizations common to all Python classes . . . 20
4.4 Packaging Ruby applications . . . 21
4.4.1 Class for Ruby setup.rb . . . 21
4.4.2 Rules for the Ruby Extras Team . . . 22
4.5 Packaging GNOME applications . . . 22
4.5.1 Class for GNOME . . . 22
4.5.2 Rules for the GNOME Team . . . 23
4.6 Packaging KDE applications . . . 23
4.7 Packaging Java applications . . . 24
4.8 Packaging Haskell applications . . . 25

5 Maintainer Tools **26**
5.1 Maintainer Mode . . . 26
5.2 Semi-automatic `debian/control` management . . . 26
5.3 Package Check Utilities . . . 27
5.3.1 Copyright Check . . . 27
5.3.2 Build Infos . . . 27
5.3.3 List Missing Files . . . 28
5.4 Upstream Source Management . . . 28

6 Hall of examples **29**
6.1 Java (with upstream-tarball + utils + debhelper) . . . 29
6.2 Ruby (with debhelper) . . . 31
6.3 GNOME (uses autotools) . . . 31
6.3.1 `gnome-panel` package . . . 31
6.3.2 `gdm3` package . . . 32
6.4 Python (with debhelper) . . . 33
6.5 Simple Makefile (with dpatch) . . . 34
6.6 Perl (with upstream-tarball + utils + debhelper) . . . 34

7 Useful tools **36**
7.1 **cdbs-edit-patch** (provided with CDBS) . . . 36

8 Conclusion **37**

List of Tables

2.1 Variables commonly available in `debian/rules` . . . 3
2.2 Targets commonly available in `debian/rules` . . . 5

3.1 Variables available in `debian/rules` with debhelper rules . . . 9
3.2 Targets available in `debian/rules` with debhelper rules . . . 9
3.3 Debhelper commands commonly managed . . . 10

4.1 Debhelper commands managed for Python packaging . . . 18
4.2 Debhelper commands managed for Ruby packaging . . . 21
4.3 Debhelper commands managed for GNOME packaging . . . 22

Foreword

This documentation describes what we succeeded to learn about CDBS usage, with as much details as possible. Nevertheless, we are not using the whole set of available features ourselves, and some parts of this documentation were written for mere convenience and completeness.

Please note some examples in this documentation contain folded content which is necessary to keep the pages at a reasonable width; take care to unfold them when necessary before using them (e.g., parts of the `debian/control` content must not be folded or build will fail or result be incorrect).

If you find mistakes or missing information, feel free to contact Marc Dequènes (Duck) duck@duckcorp.org.

Chapter 1

Introduction

1.1 A bit of history

CDBS was written by Jeff Bailey and Colin Walters in march 2003, later joined by 4 other developers.

In 2004, we (Rtp and me) were experimenting CDBS, and it was quickly obvious the lack of documentation (a very small set of examples and spare notes shipped in the package) was preventing us from using it widely in our packages. Thus, we started to write some notes on CDBS usage, quickly growing to several pages. This documentation is a revised and improved version from the original DuckCorp Wiki page.

Nowadays, CDBS is mostly maintained by Peter Eisentraut and Jonas Smedegaard (and occasionally me). Information on the project can be found on the alioth page; contributors welcome ;-).

1.2 Why CDBS ?

CDBS is designed to simplify the maintainer's work so that they only need to think about packaging and not maintaining a `debian/rules` file that keeps growing bigger and more complicated. So CDBS can handle for you most of common rules and detect some parts of your configuration.

CDBS only uses simple makefile rules and is easily extensible using classes. Classes for handling autotools buildsys, applying patches to source, GNOME softwares, Python intall, and so on are available.

CDBS advantages:

- short, readable and efficient `debian/rules`
- automates debhelper and autotools for you so you don't have to bother about this unpleasant and repetitive tasks
- maintainer can focus on real packaging problems because CDBS helps you but do not limit customization
- classes used in CDBS have been well tested so you are using error-proof rules and avoid dirty hacks to solve common problems
- switching to CDBS is easy
- can be used to generate Debian files (like `debian/control` for GNOME Team Uploaders inclusion)
- CDBS is easily extendable
- It l70>< !!!

Chapter 2

First steps

2.1 Convert a package to CDBS

Converting to CDBS is easy; A simple `debian/rules` for a C/C++ software with no extra rules would be written as this:

```
#!/usr/bin/make -f

include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/class/autotools.mk
```

No, I'm not joking, this is sufficient to handle autotools management, like updating `config.{guess|sub}`, cleanup temp files after build and launch all common debhelper stuff.

Just use compat level 7 (or lower if needed, but not lower than 4 or it may not work), create your `pkg.install`, `pkg`.info, etc as you usually do with **dh_*** commands, and CDBS would call them if necessary, auto-detecting a lot of things.

Important
If `debian/control` management is activated (see below), build dependency on `cdbs` is automatically added, if not, you will have to do it yourself.

Warning
Beware your working directory *MUST NOT* have spaces or CDBS would probably fail; see #306941

2.2 Basic settings and available variables

Most variables you can use in `debian/rules`:

You can customize basic build parameters this way:

```
DEB_SRCDIR = $(CURDIR)/src
DEB_BUILDDIR = $(DEB_SRCDIR)/build
DEB_DESTDIR = $(CURDIR)/plop/
```

Remember you can get the package directory using the `CURDIR` variable.

[1] /!\ deprecated, only defined to provide backward compatibility /!\ (see man dpkg-architecture for more information)
[2] automatically set by GNU make.
[3] see the custom rules below.

Variable	Usage
source package information	
DEB_SOURCE_PACKAGE	name of the source package
DEB_VERSION	full Debian version
DEB_NOEPOCH_VERSION	Debian version without epoch
DEB_UPSTREAM_VERSION	upstream version
DEB_ISNATIVE	non-empty if package is native
binary packages information	
DEB_ALL_PACKAGES	list of all binary packages
DEB_INDEP_PACKAGES	list of architecture independent binary packages
DEB_ARCH_PACKAGES	list of architecture dependant binary packages
DEB_UDEB_PACKAGES	list of udeb binary packages, if any
DEB_PACKAGES	list of non-udeb binary packages
DEB_INDEP_REGULAR_PACKAGES	list of non-udeb architecture independent binary packages
DEB_ARCH_REGULAR_PACKAGES	list of non-udeb architecture dependant binary packages
DEB_DBG_PACKAGES	list of debug packages
system information	
DEB_HOST_GNU_TYPE	GNU type on the host machine
DEB_HOST_GNU_SYSTEM	system part of GNU type on the host machine
DEB_HOST_GNU_CPU	CPU part of GNU type on the host machine
DEB_HOST_ARCH	Debian architecture name on the host machine
DEB_HOST_ARCH_CPU	CPU part of the Debian architecture name on the host machine
DEB_HOST_ARCH_OS	OS part of the Debian architecture name on the host machine
DEB_BUILD_GNU_TYPE	GNU type for this build
DEB_BUILD_GNU_SYSTEM	system part of GNU type for this build
DEB_BUILD_GNU_CPU	CPU part of GNU type for this build
DEB_BUILD_ARCH	Debian architecture name for this build
DEB_BUILD_ARCH_CPU	CPU part of the Debian architecture name for this build
DEB_BUILD_ARCH_OS	OS part of the Debian architecture name for this build
cdbs_crossbuild	non-empty if cross-building
DEB_ARCH[1]	old Debian architecture name
directories	
CURDIR[2]	package directory
DEB_SRCDIR	where sources are (defaults to ".")
DEB_BUILDDIR	in which directory to build (defaults to DEB_SRCDIR)
DEB_DESTDIR	in which directory to install the software (defaults to $(CURDIR)/debian/<package> if there is only one binary package or $(CURDIR)/debian/tmp/ if many)
per binary package, for '*action/package*' targets[3]	
cdbs_curpkg	name of the package this target applies to
miscellaneous	
DEB_VERBOSE_ALL	if set, ask CDBS to be more verbose

Table 2.1: Variables commonly available in debian/rules

2.3 Custom build rules

CDBS defines extra make targets, and do needed actions in these rules. You can use some of these targets as hooks to add your own customizations. The CDBS classes sometimes add new targets for a specific usage, and you can use some of them as well. Which targets are usuable as hooks is easy: they use the make "double-colon rules" feature, and most of them are described in this manual (we try to get in sync with development to document them all).

Warning
Beware to use targets *after* needed CDBS includes.

Most targets you can use in `debian/rules` (called in this order): The most useful for a majority of use cases are the *`action`*/*`package`*, post-patches, and clean targets.

2.3.1 Examples

Suppose you want custom rules for the source package foo, creating foo (arch-dep) and foo-data (arch-indep), you simply need to add some lines to `debian/rules`.

To add pre-configure actions:

```
makebuilddir/foo::
   ln -s plop plop2
```

To add post-configure actions:

```
configure/foo::
   sed -ri 's/PLOP/PLIP/' Makefile

configure/foo-data::
   touch src/z.xml
```

/!\ in this case we are talking about package configuration and NOT about a configure script made with autotools.

To add post-build actions:

```
build/foo::
   /bin/bash debian/scripts/toto.sh

build/foo-data::
   $(MAKE) helpfiles
```

To add post-install actions:

```
install/foo::
   cp debian/tmp/myfoocmd debian/foo/foocmd
   find debian/foo/ -name "CVS" -depth -exec rm -rf {} \;

install/foo-data::
   cp data/*.png debian/foo-data/usr/share/foo-data/images/
   dh_stuff -m ipot -f plop.bz3 debian/foo-data/libexec/
```

To add post deb preparation actions:

[4] -arch and -indep variants are defined too, and follow the same recipe with the corresponding -indep and -arch targets ignored
[5] see Debian Policy §4.9.
[6] if we are building in the source directory, a broken upstream build system would leave dirty things behind, and you will have to handle this yourself.

Target	Usage
targets called during 'build'[4] Debian essential target[5], in this order	
debian/control	actions after `debian/control` is generated, if you use `debian/control` management
makebuilddir/*package*	actions after build directory for package *package* has been created
makebuilddir	actions after build directories for all packages have been created
pre-build	actions before anything serious is done (like patching or configuring)
update-config	actions after sources are patched and general build system files (like autotools scripts) are updated
post-patches	actions after sources are ready to use
common-configure-arch	actions after configuration, for arch-dependent packages
common-configure-indep	actions after configuration, for arch-independent packages
configure/*package*	actions after configuration, for the package *package*
common-build-arch	actions after compilation, for arch-dependent packages
common-build-indep	actions after compilation, for arch-independent packages
build/*package*	actions after compilation, for the package *package*
targets called during 'binary'[4] Debian essential target[5], in this order	
common-post-build-arch	actions immediately after build for the arch-dependent packages
common-post-build-indep	actions immediately after build for the arch-independent packages
common-install-prehook-arch	actions before anything serious is done (like installing files), for arch-dependent packages
common-install-prehook-indep	actions before anything serious is done (like installing files), for arch-independent packages
common-install-arch	actions after installation, for arch-dependent packages
common-install-indep	actions after installation, for arch-independent packages
install/*package*	actions after installation, for the package *package*
common-binary-arch	actions after everything is installed, for arch-dependent packages
common-binary-indep	actions after everything is installed, for arch-independent packages
binary/*package*	actions after everything is installed, for the package *package*
targets called during 'clean' Debian essential target[5], in this order	
cleanbuilddir/*package*	actions after build directory for package *package* has been removed
cleanbuilddir	actions after build directories for all packages have been removed
reverse-config	actions after general build system files (like autotools scripts) are restored
clean	actions after everything is cleaned (sources are unpatched and general build system file are removed, most[6] files generated by the build process are removed)

Table 2.2: Targets commonly available in `debian/rules`

```
binary/foo::
   strip --remove-section=.comment --remove-section=.note --strip-unneeded \
      debian/foo/usr/lib/foo/totoz.so
```

To add pre-clean actions:

```
cleanbuilddir/foo::
   rm -f debian/fooman.1
```

2.4 Common Build Options

Default optimization is set using `DEB_OPT_FLAG` which default to "-O2"; you can override it. `CFLAGS` and `CXXFLAGS` are set to "-g -Wall $(DEB_OPT_FLAG)", CPPFLAGS is untouched from environment, but you can override these settings on a per-package basis using `CFLAGS_`*`package`*, `CXXFLAGS_`*`package`*, and `CPPFLAGS_`*`package`* variables.

`DEB_BUILD_OPTIONS` is a well known Debian environment variable, not a CDBS one, containing special build options (a comma-separated list of keywords). CDBS does check `DEB_BUILD_OPTIONS` to take these options into account; see details in each class.

`DEB_BUILD_PARALLEL`, if set, allow activating parallel builds in certain classes (makefile, automake, perlmodule, qmake, cmake, and scons classes use build commands supporting parallel builds). CDBS then uses `DEB_PARALLEL_JOBS` to know how many builds to launch (which defaults to `parallel=` parameter in `DEB_BUILD_OPTIONS`, but can be overridden).

Chapter 3

Common Tasks

3.1 Automatic tarball management

To use the CDBS tarball system, just add his line to your `debian/rules`, and specify the name of the top directory of the extracted tarball:

```
include /usr/share/cdbs/1/rules/tarball.mk

DEB_TAR_SRCDIR := foosoft
```

CDBS will recognize tarballs with the following extensions: .tar .tgz .tar.gz .tar.bz .tar.bz2 .tar.lzma .zip

The tarball location is auto-detected if in the top source directory, or can be specified:

```
DEB_TARBALL := $(CURDIR)/tarballdir/mygnustuff_beta-1.2.3.tar.gz
```

CDBS will handle automatic decompression and cleanups, automagically set `DEB_SRCDIR` and `DEB_BUILDDIR` for you, and integrate well with other CDBS parts (like autotools class). If `DEB_VERBOSE_ALL` is set, tarball decompression is made verbose.

Warning
The `DEB_AUTO_CLEANUP_RCS` feature, used to clean up sources from dirty SCM-specific dirs and file , has been removed since version 0.4.39; use `DH_ALWAYS_EXCLUDE` (from debhelper) instead .

Important
If needed, and if `debian/control` management is activated (see below), build dependency on `bzip2` or `unzip` is automatically added, if not, you will have to do it yourself.

3.2 Patching sources

3.2.1 Rules for simple-patchsys

First, patching sources directly is really BAD™, so you need a way to apply patches without touching any files. These rules, inspired by the Dpatch system, are quite similar and powerful. All you need is diff/patch knowledge, CDBS is doing the rest.

That's quite hard, so please listen carefully and prepare for examination.

First, add this line to your `debian/rules`:

```
include /usr/share/cdbs/1/rules/simple-patchsys.mk
```

And then use it !

Create the directory `debian/patches` and put your patches in it. Files should be named so as to reflect in which order they have to be applied, and must finish with the `.patch` or `.diff` suffix. The class would take care of patching before configure and unpatch after clean. It is possible to use patch level 0 to 3, and CDBS would try them and use the right level automatically. The system can handle compressed patch with additional `.gz` or `.bz2` suffix and uu-encoded patches with additional `.uu` suffix.

You can customize the directories where patches are searched, and the suffix like this: (defaults are: .diff .diff.gz .diff.bz2 .diff.uu .patch .patch.gz .patch.bz2 .patch.uu)

```
DEB_PATCHDIRS := debian/mypatches
DEB_PATCH_SUFFIX := .plop
```

In case of errors when applying, for example `debian/patches/01_hurd_ftbfs_pathmax.patch`, you can read the log for this patch in `debian/patches/01_hurd_ftbfs_pathmax.patch.level-0.log` ('0' because a level 0 patch).

Important
If `debian/control` management is activated (see below), build dependency on `patchutils` is automatically added, if not, you will have to do it yourself.

3.2.2 Rules for dpatch

To use Dpatch as an alternative to the CDBS included patch system, just add his line to your `debian/rules`:

```
include /usr/share/cdbs/1/rules/dpatch.mk
# needed to use the dpatch tools (like dpatch-edit-patch)
include /usr/share/dpatch/dpatch.make
```

Now you can use Dpatch as usual and CDBS would call it automatically.

Warning
You should include dpatch.mk *AFTER* autotools.mk or gnome.mk in order to have dpatch extension work correctly.

Important
If `debian/control` management is activated (see below), build dependency on `dpatch` and `patchutils` is automatically added, if not, you will have to do it yourself.

3.2.3 Rules for quilt

To use Quilt as an alternative to the CDBS included patch system, just add his line to your `debian/rules`:

```
include /usr/share/cdbs/1/rules/patchsys-quilt.mk
```

Now you can use Quilt as usual and CDBS would call it automatically.

Important
If `debian/control` management is activated (see below), build dependency on `quilt` and `patchutils` is automatically added, if not, you will have to do it yourself.

3.3 Using Debhelper

3.3.1 Not managing Debhelper

Yes, CDBS is doing almost everything for you :).

Just add this line to the beginning of your `debian/rules` file:

```
include /usr/share/cdbs/1/rules/debhelper.mk
```

In case of a missing compat information, CDBS would create a `debian/compat` file with compatibility level 7. If you are using an obsolete `DH_COMPAT` variable in your `debian/rules`, you should get rid of it. In this case, or in case you would like CDBS not to create a `debian/compat` file, you can disable this feature by setting `DEB_DH_COMPAT_DISABLE` to a non-void value.

Extra variables you can use in `debian/rules` with these rules:

Variable	Usage
debhelper	
`DH_COMPAT`	reflect the debhelper compatibility level (set by CDBS if unset)
`DH_VERBOSE`	ask debhelper to be verbose (defaults to the value of `DEB_VERBOSE_ALL`)
per binary package, for '*`action`*/*`package`*' targets	
`is_debug_package`	set if the package is a debug package

Table 3.1: Variables available in `debian/rules` with debhelper rules

Extra targets can be used in `debian/rules` with these rules (called in this order): For example, to add post install preparation

Target	Usage
targets called just before 'common-install-prehook-arch' and 'common-install-prehook-indep' common targets, in this order	
common-install-prehook-impl	actions before installation and after debhelper prepared the directories
targets called just before 'build/*`package`*' common targets, in this order	
binary-install/*`package`*	actions after the upstream build system and debhelper installed everything
binary-post-install/*`package`*	actions after everything is installed (after the 'binary-install' rule, which may be customized in a class or by yourself)
binary-strip/*`package`*	actions after stripping executables and libraries
binary-fixup/*`package`*	actions after gzipping certain files (doc, examples, ...) and fixing permissions
binary-predeb/*`package`*	actions just before creating the debs
binary-makedeb/*`package`*	actions just after creating the debs

Table 3.2: Targets available in `debian/rules` with debhelper rules

(after debhelper work):

```
binary-install/foo::
   chmod a+x debian/foo/usr/bin/pouet
```

These rules handles the following debhelper commands for each binary package automatically (with the parameters you can use to customize their behavior): Other debhelper commands can be handled in specific classes or may be called in custom

Commands	Parameters	
commands called during 'common-install-prehook-impl rule, in this order		
dh_clean[1]	DEB_CLEAN_EXCLUDE	regular expressions matching files which should not be cleaned
dh_prep[2]	DEB_CLEAN_EXCLUDE	regular expressions matching files which should not be cleaned
dh_installdirs	DEB_INSTALL_DIRS_ALL	subdirectories to create in installation staging directory, for all packages
commands called during 'install/*package*' rules, in this order		
dh_installdirs	DEB_INSTALL_DIRS_*package*	subdirectories to create in installation staging directory, for package *package*
commands called during 'binary-install/*package*' rules, in this order		
dh_installdocs	DEB_INSTALL_DOCS_ALL	files which should go in /usr/share/doc/package, for all packages
	DEB_INSTALL_DOCS_*package*	files which should go in /usr/share/doc/package, for package *package*
dh_installexamples	DEB_INSTALL_EXAMPLES_*package*	files which should go in /usr/share/doc/package/examples, for package *package*
dh_installman	DEB_INSTALL_MANPAGES_*package*	manpages to install in the package, for package *package*
dh_installinfo	DEB_INSTALL_INFO_*package*	info files which should go in /usr/share/info, for package *package*
dh_installmenu	DEB_DH_INSTALL_MENU_ARGS	extra arguments passed to the command
dh_installcron	DEB_DH_INSTALL_CRON_ARGS	extra arguments passed to the command
dh_installinit	DEB_UPDATE_RCD_PARAMS_*package*	arguments passed to update-rc.d in init scripts, for package *package*
	DEB_UPDATE_RCD_PARAMS	arguments passed to update-rc.d in init scripts, for all packages (overrides DEB_UPDATE_RCD_PARAMS_*package* parameters)
	DEB_DH_INSTALLINIT_ARGS	extra arguments passed to the command
dh_installdebconf	DEB_DH_INSTALLDEBCONF_ARGS	extra arguments passed to the command
dh_installemacsen	DEB_EMACS_PRIORITY	overrides the default priority for the site-start.d file
	DEB_EMACS_FLAVOR	overrides the default Emacs flavor for the site-start.d file
	DEB_DH_INSTALLEMACSEN_ARGS	extra arguments passed to the command
dh_installcatalogs	DEB_DH_INSTALLCATALOGS_ARGS	extra arguments passed to the command
dh_installpam	DEB_DH_INSTALLPAM_ARGS	extra arguments passed to the command
dh_installlogrotate	DEB_DH_INSTALLLOGROTATE_ARGS	extra arguments passed to the command
dh_installlogcheck	DEB_DH_INSTALLLOGCHECK_ARGS	extra arguments passed to the command
dh_installchangelogs	DEB_INSTALL_CHANGELOGS_ALL	files which should be interpreted as upstream changelog, for all packages
	DEB_INSTALL_CHANGELOGS_*package*	files which should be interpreted as upstream changelog, for package *package*
	DEB_DH_INSTALLCHANGELOGS_ARGS	extra arguments passed to the command

rules.

Important
If `debian/control` management is activated (see below), build dependency on `debhelper` is automatically added, if not, you will have to do it yourself.
Having a versioned dependency on `debhelper` is recommended as it will ensure people will use the version providing the necessary features (CDBS `debian/control` management will do it).

3.3.2 Debhelper customization examples

To specify a tight dependency on a package containing shared libraries:

```
DEB_DH_MAKESHLIBS_ARGS_libfoo := -V"libfoo (>= 0.1.2-3)"
DEB_SHLIBDEPS_LIBRARY_arkrpg := libfoo
DEB_SHLIBDEPS_INCLUDE_arkrpg := debian/libfoo/usr/lib/
```

To install a changelog file with an uncommon name as `ProjectChanges.txt.gz`:

```
DEB_INSTALL_CHANGELOGS_ALL := ProjectChanges.txt
```

To avoid compressing files with `i.py` extension:

```
DEB_COMPRESS_EXCLUDE := .py
```

To register a debug library package libfoo-dbg for libfoo (which needs unstripped `.so`) in compat mode 4:

```
DEB_DH_STRIP_ARGS := --dbg-package=libfoo
```

Starting from compat mode 5, CDBS automatically detect -dbg packages and pass the needed arguments to **dh_strip**; `DEB_DH_STRIP_ARGS` can still be useful to pass additional parameters like excluded items (`--exclude=item`).

Perl-specific debhelper options (**dh_perl** call is always performed):

```
# Add a space-separated list of paths to search for perl modules
DEB_PERL_INCLUDE := /usr/lib/perl-z
# Like the above, but for the 'libperl-stuff' package
DEB_PERL_INCLUDE_libperl-stuff := /usr/lib/perl-plop

# Overrides options passed to dh_perl
DEB_DH_PERL_ARGS := -d
```

To avoid loosing temporary generated files in **dh_clean** processing (rarely useful):

```
# files containing these pattern would not be deleted
# (beware CDBS~changelog has a typo while highlighting new DEB_CLEAN_EXCLUDE*S* feature)
DEB_CLEAN_EXCLUDE := precious keep
```

[1] for compat level <7.
[2] for compat level >=7.
[3] in compat level >=7, debhelper automatically looks for files in `debian/tmp`.
[4] not called for debug packages.
[5] generally not needed, unless your package builds multiple flavors of the same library.
[6] generally not needed, unless your package builds multiple flavors of the same library or the library is installed into a directory not in the regular library search path.

3.4 Cross-Building

Since version 0.4.83, CDBS has a working cross-building support. If cross-building, a few adaptations are automatically made:

- the `cdbs_crossbuild` variable is automatically set (and can be used in `debian/rules`)
- in the makefile class: CC is passed to the make calls with the value "$(`DEB_HOST_GNU_TYPE`)-gcc"
- in the autotools class: CC and CXX are only passed to the configure call if explicitly set in environment or `debian/rules`

Chapter 4

Specific Tasks

4.1 Packaging applications with a generic build-system

4.1.1 Class for Makefile

This class is for the guys who only have a Makefile (no autotools available) to build the program. You only need to have four rules in the Makefile:

- one for cleaning the build directory (i.e. mrproper)
- one for building your software (i.e. myprog)
- one for checking if the software is working properly (i.e. check)
- one for installing your software (i.e. install)

To be honest, the install rules is not a must-have, but it always helps a lot when you've got it.

4.1.1.1 Classic Build

The first operation, is to write the `debian/rules`. First, we add the include lines:

```
include /usr/share/cdbs/1/class/makefile.mk
```

Now, it remains to tell cdbs the name of our four Makefile rules. For the previous examples it gives:

```
DEB_MAKE_CLEAN_TARGET    := mrproper
# if you detect authors's loss of sanity, tell CDBS not to try running the nonexistent  ↩
    clean rule, and do the job yourself in debian/rules
DEB_MAKE_CLEAN_TARGET    :=
DEB_MAKE_BUILD_TARGET    := myprog
DEB_MAKE_INSTALL_TARGET  := install DESTDIR=$(CURDIR)/debian/tmp/
# no check for this software
DEB_MAKE_CHECK_TARGET :=

# allow changing the makefile filename in case of emergency exotic practices
DEB_MAKE_MAKEFILE        := MaKeFiLe
# example when changing environment variables is necessary:
DEB_MAKE_ENVVARS    := CFLAGS="-fomit-frame-pointer"
```

`DEB_BUILD_OPTIONS` is checked for the following options:

- noopt: use -O0 instead of -O2
- nocheck: skip the check rule

If your Makefile doesn't support the `DESTDIR` variable, take a look in it and find the variable responsible for setting installation directory. If you don't find some variable to do this, you'll have to patch the file...

That's all :).

4.1.1.2 Multiple "flavors" Build

After setting the basic parameters like for a classic build, you just need to specify the list of your "flavors" and ensure you've got a specific build directory (which can't be the same as the source directory or all builds would walk over themselves in a big mess):

```
# needs to be declared before including makefile.mk
DEB_MAKE_FLAVORS = full nox light
DEB_BUILDDIR = build
```

If you need to create customized rules like this:

```
$(cdbs_make_clean_nonstamps)::
  $(if $(cdbs_make_flavors),rm -f extra-stuff-$(cdbs_make_curflavor))

$(cdbs_make_install_stamps)::
  $(if $(cdbs_make_flavors),dostuff --with=$(cdbs_make_curflavor))
```

you can use these extra CDBS variables:

- `cdbs_make_flavors`: sorted flavor list (nonempty if multibuild activated)
- `cdbs_make_curflavor`: current flavor for this build
- `cdbs_make_curbuilddir`: build directory for the current flavor
- `cdbs_make_curdestdir`: installation directory for the current flavor

4.1.2 Class for Autotools

This class is able to use configure scripts and makefiles generated with autotools (and possibly libtool). All rules are called automatically and clean rules to remove generated files during build are also added. This class in fact improves the makefile class to support autotools features and provide good defaults.

4.1.2.1 Classic Build

To use it, just add this line to your `debian/rules`

```
include /usr/share/cdbs/1/class/autotools.mk
```

CDBS automatically handles common flags to pass to the configure script, but it is possible to give some extra parameters:

```
DEB_CONFIGURE_EXTRA_FLAGS := --with-ipv6 --with-foo
```

If the build system uses non-standard configure options you can override CDBS default behavior:

```
COMMON_CONFIGURE_FLAGS := --program-dir=/usr
```

(notice that `DEB_CONFIGURE_EXTRA_FLAGS` would still be appended)

If some specific environment variables need to be setup, use:

```
DEB_CONFIGURE_SCRIPT_ENV += BUILDOPT="someopt"
```

Warning
Prefer use of += instead of := not to override other environment variables (CC / CXX / CFLAGS / CXXFLAGS / CPPFLAGS / LDFLAGS) propagated in the CDBS default.

CDBS will automatically update `config.sub`, `config.guess`, and `config.rpath` before build and restore the old ones at clean stage (even if using the tarball system). If needed, and if `debian/control` management is activated, `autotools-dev` and/or `gnulib` will then be automatically added to the build dependencies (needed to find updated versions of the files).

If the program does not use the top source directory to store autoconf files, you can teach CDBS where it is to be found:

```
DEB_AC_AUX_DIR = $(DEB_SRCDIR)/autoconf
```

CDBS can be asked to update libtool, autoconf, and automake files, but this behavior is likely to break the build system and is *STRONGLY* discouraged. Nevertheless, if you still want this feature, set the following variables:

- `DEB_AUTO_UPDATE_LIBTOOL`: `pre` to call libtoolize, or `post` to copy system-wide **libtool** after configure is done
- `DEB_AUTO_UPDATE_ACLOCAL`: **aclocal** version to use
- `DEB_AUTO_UPDATE_AUTOCONF`: **autoconf** version to use
- `DEB_AUTO_UPDATE_AUTOHEADER`: **autoheader** version to use
- `DEB_AUTO_UPDATE_AUTOMAKE`: **automake** version to use
- `DEB_ACLOCAL_ARGS`: extra arguments to **aclocal** call (defaults to `DEB_SRCDIR/m4` if exists)
- `DEB_AUTOMAKE_ARGS`: extra arguments to **automake** call

(corresponding build dependencies will automatically be added)

The following make parameters can be overridden:

```
# these are the defaults CDBS provides
DEB_MAKE_INSTALL_TARGET := install DESTDIR=$(DEB_DESTDIR)
DEB_MAKE_CLEAN_TARGET := distclean
DEB_MAKE_CHECK_TARGET :=

# example to work around dirty makefile
DEB_MAKE_INSTALL_TARGET := install prefix=$(CURDIR)/debian/tmp/usr

# example with nonexistent install rule for make
DEB_MAKE_INSTALL_TARGET :=

# example to activate check rule
DEB_MAKE_CHECK_TARGET := check

# overriding make-only environment variables :
# (should never be necessary in a clean build system)
# (example borrowed from the bioapi package)
DEB_MAKE_ENVVARS    := "SKIPCONFIG=true"
```

`DEB_BUILD_OPTIONS` is checked for the following options:

- noopt: use -O0 instead of -O2

- nocheck: skip the check rule

If you are using CDBS version < 0.4.39, it automatically cleans autotools files generated during build (`config.cache`, `config.log`, and `config.status`). Since version 0.4.39, CDBS leave them all considering it is not his job to correct an upstream buildsys misbehavior (but you may remove them in the clean rule if necessary before you get the issue solved by authors).

4.1.2.2 Multiple "flavors" Build

The basics are the same as in the makefile class (on which the autotools class depends). Combined with this class you can do interesting customizations like this:

```
# global configure options
DEB_CONFIGURE_EXTRA_FLAGS = --enable-gnutls

# per flavor configure options
DEB_CONFIGURE_EXTRA_FLAGS_full = --enable-gtk --enable-pam
DEB_CONFIGURE_EXTRA_FLAGS_nox = --enable-pam
DEB_CONFIGURE_EXTRA_FLAGS_light = --disable-unicode
DEB_CONFIGURE_EXTRA_FLAGS += $(DEB_CONFIGURE_EXTRA_FLAGS_$(cdbs_make_curflavor))
```

4.1.3 Class for CMake

This class is intended to handle build systems using CMake.

To use this class, add this line to your `debian/rules`:

```
include /usr/share/cdbs/1/class/cmake.mk
```

This class is an extension of the Makefile class, calling **cmake** in `DEB_SRCDIR` with classic parameters and then calling **make**. In the call to **cmake**, the install prefix and path to CC/CXX as well as CFLAGS/CXXFLAGS are given automatically. You can pass extra arguments using `DEB_CMAKE_EXTRA_FLAGS`, and in strange cases where you need a special configuration you can override the default arguments using `DEB_CMAKE_NORMAL_ARGS`.

4.1.4 Class for qmake

This class is intended to handle build systems using qmake, mostly used to build Qt applications.

To use this class, add this line to your `debian/rules`:

```
include /usr/share/cdbs/1/class/qmake.mk
```

This class is an extension of the Makefile class, calling **qmake** in `DEB_BUILDDIR` with classic parameters and then calling **make**. In the call to **qmake**, the path to CC/CXX as well as CPPFLAGS/CFLAGS/CPPFLAGS/CXXFLAGS are given automatically. Configuration options can be given using `DEB_QMAKE_CONFIG_VAL` (resulting in adding content to `CONFIG`), the later defaulting to `nostrip` if this option is set in `DEB_BUILD_OPTIONS`.

4.1.5 Class for SCons

This class is intended to handle build systems using SCons, which is an alternative to make.

To use this class, add this line to your `debian/rules`:

```
include /usr/share/cdbs/1/class/scons.mk
```

The class takes care to invoke SCons and cleanup things afterwards. You can setup a few parameters:

```
# options for ALL targets (in addition to the DEB_SCONS_*_OPTIONS variables)
DEB_SCONS_OPTIONS = useALSA=1

# you can override the default build target (empty)
DEB_SCONS_BUILD_TARGET =
# options for the build target
DEB_SCONS_BUILD_OPTIONS = useGettext=1

# you can override the default install target ("install")
DEB_SCONS_INSTALL_TARGET = install devel-install
# options for the install target
DEB_SCONS_INSTALL_OPTIONS = instdir=$(CURDIR)/debian/tmp

# you can override the default clean target (.)
DEB_SCONS_CLEAN_TARGET = fullclean

# activate the check target if set to the target name
DEB_SCONS_CHECK_TARGET = testsuite
```

4.2 Packaging Perl applications

4.2.1 Subclass for Makefile (ExtUtils::MakeMaker)

Warning
This class replaces the 'perlmodule' class, which is deprecated. This one is very similar: rules and parameters are almost the same, and only the `DEB_MAKEMAKER_PACKAGE` variable is deprecated, so the conversion should be pretty easy.

This class can manage Perl builds using ExtUtils::MakeMaker (Makefile.PL) automatically.

To use this class, add this line to your `debian/rules`:

```
include /usr/share/cdbs/1/class/perl-makemaker.mk
```

You can customize build options like this:

```
# add custom MakeMaker options
DEB_MAKEMAKER_USER_FLAGS = --with-ipv6
```

As this class is a subclass of the makefile class, you can override the `DEB_MAKE_*` variables, but it should not be needed. Also beware `DEB_BUILDDIR` must match `DEB_SRCDIR` for this build system.

4.2.2 Class for Module::Build

This class can manage Perl builds using Module::Build (Build.PL) automatically.

The following parameters can be overridden:

```
# these are the defaults CDBS provides
DEB_PERL_BUILD_TARGET = build
DEB_PERL_CHECK_TARGET = test
DEB_PERL_INSTALL_TARGET = install
DEB_PERL_CLEAN_TARGET = realclean

# add your custom parameters if the defaults do not suits you
DEB_PERL_CHECK_FLAGS = fulltestsuite=1
DEB_PERL_CONFIGURE_FLAGS = destdir=$(CURDIR)/build/stuff
```

Also beware `DEB_BUILDDIR` must match `DEB_SRCDIR` for this build system.

4.2.3 Customizations common to all Perl classes

If you included the debhelper class, it can take care of calling **dh_perl**. Have a look at Perl-specific debhelper options described here.

Extra variables you can use in your `debian/rules`:

- `DEB_PERL_PACKAGES`: list of Perlpackages
- `DEB_PERL_ARCH_PACKAGES`: list of architecture dependent Perlpackages
- `DEB_PERL_INDEP_PACKAGES`: list of architecture independent Perlpackages

Important
If `debian/control` management is activated (see below), build dependency on `perl` is automatically added, if not, you will have to do it yourself.

4.3 Packaging Python applications

With the new policy all versioned packages (python*ver*-*app*) are collapsed into a single package (python-*app*). The Python classes are able to move python scripts and .so files in the new locations automatically. You can use the auto control file generation feature to ensure your Build-Depends are set correctly for the new needed tools. The classes also remove `.egg-info` directories sometimes left over by Python distutils.

Before choosing the right class to use (depending on the build system used upstream, and described in the next sections), you need to choose a Python package manager. You may wonder why this choice at all, and why Debian Python folks did not stick to a single common tool, but this is a long tragedy, and you'd find most of it on the Debian mailing-lists, if you're curious. You can find more information on these tools on this page and in the corresponding packages.

Then, add the following *before* the Python class inclusion to your `debian/rules`:

```
# select the python system you want to use : pysupport or pycentral
# (this MUST be done before including the class)
DEB_PYTHON_SYSTEM = pysupport
```

It can handle the following debhelper commands automagically:

Commands	Parameters	
commands called during 'binary-post-install/*package*' rules, in this order		
dh_pysupport or **dh_pycentral**.	`DEB_PYTHON_PRIVATE_MODULES_DIRS`[1]	list of private modules directories, for all packages
	`DEB_PYTHON_PRIVATE_MODULES_DIRS_`*package*[1]	list of private modules directories, for package *package*

Table 4.1: Debhelper commands managed for Python packaging

[1] needed to automatically handle bytecompilation for these modules.

4.3.1 Class for Python distutils

This class can manage common Python builds using distutils automatically.

Warning
Since 0.4.53, this class does not handle old-policy packages (pre-Etch) anymore.

To use this class, add these lines to your `debian/rules`:

```
# Don't forget to declare DEB_PYTHON_SYSTEM before this line
include /usr/share/cdbs/1/class/python-distutils.mk
```

You can customize build options like this:

```
############## These variables have not changed (same in the old and new policy)

# change the Python build script name (default is 'setup.py')
DEB_PYTHON_SETUP_CMD := install.py

# clean options for the Python build script
DEB_PYTHON_CLEAN_ARGS = -all

# build options for the Python build script
DEB_PYTHON_BUILD_ARGS = --build-base="$(DEB_BUILDDIR)/specific-build-dir"

# common additional install options for all binary packages
#   ('--root' option is always set)
DEB_PYTHON_INSTALL_ARGS_ALL = --no-compile --optimize --force
```

Complete `debian/rules` example using python-support for a module (editobj):

```
#!/usr/bin/make -f
# -*- mode: makefile; coding: utf-8 -*-

include /usr/share/cdbs/1/rules/debhelper.mk
DEB_PYTHON_SYSTEM = pysupport
include /usr/share/cdbs/1/class/python-distutils.mk
include /usr/share/cdbs/1/rules/patchsys-quilt.mk


DEB_COMPRESS_EXCLUDE := .py


$(patsubst %,install/%,$(cdbs_python_packages))::
        mv debian/$(cdbs_curpkg)/usr/lib/python*/site-packages/editobj/icons \
                debian/$(cdbs_curpkg)/usr/share/$(cdbs_curpkg)

$(patsubst %,binary-install/%,$(cdbs_python_packages))::
        find debian/$(cdbs_curpkg)/usr/share/ -type f -exec chmod -R a-x {} \;
```

4.3.2 Subclass for autotools

For a Python module using autotools, it may be convenient to use this class, which is a subclass of the generic autotools class. It currently complement the `DEB_CONFIGURE_SCRIPT_ENV` variable with the `PYTHON` parameter, and adds needed build dependencies if `debian/control` management is activated. Moreover, if the package is to be built for multiple Python

versions, then the multiple "flavors" build is automatically activated (DEB_MAKE_FLAVORS is set accordingly, but still can be overridden is needed).

To use this class, add this line to your debian/rules:

```
include /usr/share/cdbs/1/class/python-autotools.mk
```

4.3.3 Class for Python Sugar

This class can manage activity extensions for the Sugar Learning Platform (originally developed for the One Laptop per Child XO-1 netbook) automatically.

To use this class, add these lines to your debian/rules:

```
include /usr/share/cdbs/1/class/python-sugar.mk

# Compulsory: Sugar branches supported by this activity extension
DEB_SUGAR_BRANCHES = 0.84 0.86
```

4.3.4 Customizations common to all Python classes

You can customize build options like this:

```
############## These variables are additions for the new policy

# packages holding the collapsed content of all supported versions
# CDBS defaults to the first non -doc/-dev/-common package listed in "debian/control"
# this variable allows to override automatic detection
DEB_PYTHON_MODULE_PACKAGES = mypyapp

# list of private modules directories (for all binary packages) (needed to automatically  ↩
    handle bytecompilation)
DEB_PYTHON_PRIVATE_MODULES_DIRS = /usr/share/mypyapp/my-pv-module
# or for a specific binary package
DEB_PYTHON_PRIVATE_MODULES_DIRS_python-stuff-client = /usr/share/mypyapp/client-pv-module

# overrides the default Python installation root directory
DEB_PYTHON_DESTDIR = $(CURDIR)/debian/python-stuff
```

You may use some read-only (meaning you MUST NOT alter them) variables in your debian/rules:

- cdbs_python_current_version: current python version number (defined only if selected package is a module)
- cdbs_python_build_versions: list of space separated version numbers for which the selected module/extension is gonna be built
- cdbs_python_destdir: Python installation root directory (works even when you didn't used DEB_PYTHON_DESTDIR
- cdbs_python_packages: list of all Python packages
- cdbs_python_arch_packages: list of all Python architecture dependent packages
- cdbs_python_indep_packages: list of all Python architecture independent packages

Warning
Do not use the `DEB_PYTHON_MODULE_PACKAGE` variable anymore, it has been obsoleted. Use the `DEB_PYTHON_MODULE_PACKAGES` to force the list of module packages, or `cdbs_python_packages` variable in rules generation.
Do not use the `cdbs_python_support_path` and `cdbs_python_module_arch` variables anymore, they have been obsoleted. `cdbs_python_support_path` is not useful anymore, just install files in the standard way and python-support would do the rest (messing into python-support internals was only a workaround when it was incomplete). `cdbs_python_module_arch` can easily be replace by a rule rewrite or a membership test using `cdbs_python_arch_packages` or `cdbs_python_indep_packages`.

4.4 Packaging Ruby applications

4.4.1 Class for Ruby setup.rb

This class can manage common setup.rb installer automatically.

To use this class, install the `ruby-pkg-tools` package, and add this line to your `debian/rules`:

```
include /usr/share/ruby-pkg-tools/1/class/ruby-setup-rb.mk
```

It can handle the following debhelper commands automagically:

Commands	Parameters	
commands called during 'binary-install/*package*' rules, in this order		
dh_rdoc[2]		

Table 4.2: Debhelper commands managed for Ruby packaging

Most ruby packages are architecture all, and then don't need being build for multiple ruby versions; your package should then be called 'lib*foo*-ruby' or '*foo*' and CDBS would automatically use the current Debian ruby version to build it. If your package contains a compiled part or a binding to an external lib, then you will have packages named 'lib*foo*-ruby1.6', 'lib*foo*-ruby1.8', and so on, then CDBS would automatically build each package with the corresponding ruby version. In this case, don't forget to add a 'lib*foo*-ruby' convenience dummy package depending on the current Debian ruby version. If you have documentation you want split into a separate package, then call it 'lib*foo*-ruby-doc'. If this is Rdoc documentation, you may want to include the debhelper class, as explained before, to have it generated and installed automagically.

You can customize build options like this:

```
# force using a specific ruby version for build
# (should not be necessary)
DEB_RUBY_VERSIONS := 1.9

# use ancestor
DEB_RUBY_SETUP_CMD := install.rb

# config options for the ruby build script
# (older setup.rb used --site-ruby instead of --siteruby)
DEB_RUBY_CONFIG_ARGS = --site-ruby=/usr/lib/ruby/1.9-beta
```

[2] generate and install Rdoc documentation

4.4.2 Rules for the Ruby Extras Team

Warning
The uploaders rule is *deprecated*. It has been decided Uploaders should only be people who made significant changes to the package (and then added manually). Read this thread for more information.

If you are part of the Ruby Extras Team, or having the Team as Uploaders, and you feel bored maintaining the list of developers, this rule is made for you.

To use this class, install the `ruby-pkg-tools` package, and add this line to your `debian/rules`:

```
include /usr/share/ruby-pkg-tools/1/rules/uploaders.mk
```

Rename your `debian/control` file to `debian/control.in` and run the clean rule (**./debian/rules clean**) to regenerate the `debian/control` file, replacing the `@RUBY_EXTRAS_TEAM@` (`@RUBY_TEAM@` is deprecated) tag with the list of developers automatically.

4.5 Packaging GNOME applications

4.5.1 Class for GNOME

Important
This class is intended to make packaging GNOME software easily, managing specific actions for you. This documentation describe the GNOME class, which is shipped with CDBS, but if you intend to work on this area seriously, you should also have a look at the GNOME Team rules in the next chapter.

This class adds a make environment variable : GCONF_DISABLE_MAKEFILE_SCHEMA_INSTALL = 1 ("This is necessary because the Gconf schemas have to be registered at install time. In the case of packaging, this registration cannot be done when building the package, so this variable disable schema registration in **make install**. This procedure if deferred until gconftool-2 is called in `debian/postinst` to register them, and in `debian/prerm` to unregister them. The **dh_gconf** script is able to add the right rules automatically for you.")

It can handle the following debhelper commands automagically:

Commands	Parameters	
commands called during 'binary-install/*package*' rules, in this order		
dh_scrollkeeper	`DEB_DH_SCROLLKEEPER_ARGS`	extra arguments passed to the command
dh_gconf	`DEB_DH_GCONF_ARGS`	extra arguments passed to the command
dh_icons	`DEB_DH_ICONS_ARGS`	extra arguments passed to the command

Table 4.3: Debhelper commands managed for GNOME packaging

Moreover it adds some more clean rules to remove:

- intltool generated files
- scrollkeeper generated files (left over `.omf.out` files in `doc` and `help` directories)

To use it, just add this line to your `debian/rules`, after the debhelper class include:

```
include /usr/share/cdbs/1/class/gnome.mk
```

For more information on GNOME specific packaging rules, look at the Debian GNOME packaging policy.

4.5.2 Rules for the GNOME Team

If you install the extra `gnome-pkg-tools` package, you will have extra rules and tools used by the GNOME Team for their packages. You should really have a look to the package documentation for additional information (mostly in `/usr/share/doc/gnome-pkg-tools/README.Debian.gz`).

If you are part of the GNOME Team, or having the Team as Uploaders, and you feel bored maintaining the list of uploaders, add this line to your `debian/rules`:

```
include /usr/share/gnome-pkg-tools/1/rules/uploaders.mk
```

Rename your `debian/control` file to `debian/control.in` and run the clean rule (**./debian/rules clean**) to regenerate the `debian/control` file, replacing the '@GNOME_TEAM@' tag with the list of developers automatically.

Warning
If you are using the `debian/control` file management, please note this rule will override this feature To cope with this problem, allowing at least Build-Depends handling, use the following work-around (until it is solved in a proper way):

```
ifneq (, $(DEB_AUTO_UPDATE_DEBIAN_CONTROL))
clean::
  sed -i "s/@cdbs@/$(CDBS_BUILD_DEPENDS)/g" debian/control
endif
```

4.6 Packaging KDE applications

Warning
This class is intended for KDE 3 only. If you want to package KDE 4 components, please refer to the `pkg-kde-tools` package instead, which is maintained by the KDE Team (and especially read the notes in `/usr/share/doc/pkg-kde-tools/README.Debian`); we should describe it in this documentation later.

To use this class, add this line to your `debian/rules` file:

```
include /usr/share/cdbs/1/class/kde.mk
```

CDBS automatically exports the following variables with the right value:

- kde_cgidir (/usr/lib/cgi-bin)
- kde_confdir (/etc/kde3)
- kde_htmldir (/usr/share/doc/kde/HTML)

`DEB_BUILDDIR`, `DEB_AC_AUX_DIR` and `DEB_CONFIGURE_INCLUDEDIR` are set to KDE defaults.

The following files are excluded from compression:

- .dcl

- .docbook
- -license
- .tag
- .sty
- .el

(take care of them if you override the DEB_COMPRESS_EXCLUDE variable)

It can handle configure options specific to KDE (not forgetting disabling rpath and activating xinerama), set the correct autotools directory, and launch make rules adequately.

You can enable APIDOX build by setting the DEB_KDE_APIDOX variable to a non-void value.

you can enable the final mode build by setting DEB_KDE_ENABLE_FINAL variable to a non-void value.

DEB_BUILD_OPTIONS is checked for the following options:

- noopt: disable optimisations (and KDE final mode, overriding DEB_KDE_ENABLE_FINAL)
- nostrip: enable KDE debug (and disable KDE final mode, overriding DEB_KDE_ENABLE_FINAL)

You can prepare the build using the 'buildprep' convenience target: **fakeroot debian/rules buildprep** (which is in fact calling the dist target of admin/Makefile.common).

4.7 Packaging Java applications

This class allows packaging Java applications using Ant (a Java-based build tool).

To use this class, add this include to your debian/rules and set the following variables:

```
include /usr/share/cdbs/1/class/ant.mk

# Set either a single (JAVA_HOME) or multiple (JAVA_HOME_DIRS) java locations
JAVA_HOME := /usr/lib/kaffe
# or set JAVACMD if you don't use default '<JAVA_HOME>/bin/java' path
#JAVACMD := /usr/bin/java

# Set Ant location
ANT_HOME := /usr/share/ant-cvs
```

You may add additional JARs like in the following example:

```
# list of additional JAR files ('.jar' extension may be omitted)
#   (path must be absolute of relative to '/usr/share/java')
DEB_JARS := /usr/lib/java-bonus/ldap-connector adml-adapter.jar
```

The property file defaults to debian/ant.properties.

You can provide additional JVM arguments using ANT_OPTS. You can provide as well additional Ant command line arguments using ANT_ARGS (global) and/or ANT_ARGS_*pkg* (for package *pkg*), thus overriding the settings in build.xml and the property file.

CDBS will build and clean using defaults target from build.xml. To override these rules, or run the install / check rules, set the following variables to your needs:

```
# override build and clean target
DEB_ANT_BUILD_TARGET = makeitrule
DEB_ANT_CLEAN_TARGET = super-clean
# i want install and test rules to be run
DEB_ANT_INSTALL_TARGET = install-all
DEB_ANT_CHECK_TARGET = check

# extra settings
DEB_ANT_BUILDFILE = debian/build.xml
# defaults to $(CURDIR)/debian/ant.properties
DEB_ANT_PROPERTYFILE = src/ant.properties
DEB_ANT_ARGS = -Dpackage=awgraphapplet -Dversion=$(DEB_NOEPOCH_VERSION)
DEB_ANT_COMPILER := org.eclipse.jdt.core.JDTCompilerAdapter
```

`DEB_BUILD_OPTIONS` is checked for the following options:

- noopt: set 'compile.optimize' Ant option to false

You should be able to fetch some more information on this java-based build tool in the Ant Apache web site.

4.8 Packaging Haskell applications

This class allows packaging Haskell applications using HBuild (the Haskell mini-distutils). You should be able to fetch some more information on Haskell distutils in this thread.

CDBS can take care of -hugs and -ghc packages: invoke `Setup.lhs` properly for clean and install part.

To use this class, add this line to your `debian/rules`:

```
include /usr/share/cdbs/1/class/hbuild.mk
```

Chapter 5

Maintainer Tools

These are specials rules which can be activated by the Maintainer to do extra checks or help automatize boring work, but having no direct link with the software build itself.

5.1 Maintainer Mode

This mode aims at activating features which are not suitable for automatic builds, but can help maintainers while they work on their package. If `DEB_MAINTAINER_MODE` is set, this mode in activated and do the following:

- activate semi-automatic `debian/control` management (`DEB_AUTO_UPDATE_DEBIAN_CONTROL`)
- activate a strict copyright check (if the utils rules are activated)

5.2 Semi-automatic `debian/control` management

Caution
Automatic `debian/control` generation using any tool is permitted into Debian as long as it is triggered manually by the developer and the latter checks the result carefully.
Autogenerating `debian/control` without any human intervention could be harmful in some ways detailed in #311724. This is not allowed in Debian.
We then urge you to avoid using `DEB_AUTO_UPDATE_DEBIAN_CONTROL` directly and instead invoke the autogeneration rules manually after you modified `debian/control.in` (this way users or buildds wouldn't have different Build-Depends when building, avoiding many problems). Do not forget to proofread the result before any upload.
Manual `debian/control` regeneration:

```
DEB_AUTO_UPDATE_DEBIAN_CONTROL=yes fakeroot debian/rules clean
```

This feature allow:

- CDBS to automatically manage some build-related Build-Depends automatically
- use of embedded shell commands
- use of CPU and System criterias to specify architecture (*EXPERIMENTAL*)

Build-related Build-Depends are dependencies introduced by the use of certain CDBS features, or auto-detected needs.

Embedded shell commands allows including hacks like:

```
Build-Depends: libgpm-dev [‘type-handling any linux-gnu‘]
```

CPU and System criterias implements support for Cpu/System fields, as a replacement for the Architecture field (which is to be implemented in dpkg in the long term, but still *EXPERIMENTAL*). Here is an example, before:

```
Architecture: all
```

and after:

```
Cpu: all
System: all
```

If these fields are used, it is also possible to include special tags to easily take advantage of the `type-handling` tool, like in this example:

```
Build-Depends: @cdbs@, procps [system: linux], plop [cpu: s390]
```

(look at the `type-handling` package documentation, for more information)

You can also change the way build-dependencies are separated inside the Build-Depends field using the `CDBS_BUILD_DEPENDS_DELIMITER` variable. It defaults recommended multiline style (see Debian Policy §7.1), but can be changed using this variable, or get back to the old default (single line coma-separated list) if unset.

Here is the recipe:

1. Rename `debian/control` into `debian/control.in`.
2. Replace `cdbs` / `debhelper` / ... Build-Depends with @cdbs@ in your `debian/control.in` like this:

   ```
   Build-Depends-Indep: @cdbs@, python-dev (>= 2.3), python-soya (>= 0.9), \
      python-soya (<< 0.10), python-openal(>= 0.1.4-4), gettext
   ```

3. Then manually (re)generate `debian/control` as explained above (see the caution part).

5.3 Package Check Utilities

To use these utilities, add this line to your `debian/rules`:

```
include /usr/share/cdbs/1/rules/utils.mk
```

5.3.1 Copyright Check

This rule uses **licensecheck** (provided by `devscripts`) to check the `debian/copyright` file.

You need to create the empty file `debian/copyright_hints` (with touch) to activate this feature and get a report at the beginning of the build. If you set `DEB_COPYRIGHT_CHECK_STRICT`, the build will fail if it could not be run or problems were found by **licensecheck**.

You can customize the **licensecheck** parameters using `DEB_COPYRIGHT_CHECK_ARGS`, and the files to check for copyright information using `DEB_COPYRIGHT_CHECK_REGEX` and `DEB_COPYRIGHT_CHECK_IGNORE_REGEX`.

5.3.2 Build Infos

This rule uses **dh_buildinfo** (provided by `dh-buildinfo`), a tool to track package versions used to build a package.

It will generate a `debian/buildinfo.gz` file, which will be automatically installed in `/usr/share/doc/package`.

5.3.3 List Missing Files

This rule is intended to avoid missing new files in a new software version, to allow the maintainer to check the list of not-installed files. This is useful when multiple binary packages are created and things may be lost in `debian/tmp`.

The rule must be called manually, *after a build with the* `-nc` ***dpkg-buildpackage*** *option* (to avoid cleaning the `debian` directory):

```
fakeroot debian/rules list-missing
```

It generates a list of installed files `debian/cdbs-install-list` found in `debian/tmp` and a list of files installed in the package (or ignored) `debian/cdbs-package-list`, then display the ones missing in the package. You can add a list of files to ignored, because you want to remember you wanted them excluded, in `debian/not-installed`. If you set `DEB_BUILDINFO_STRICT`, the build will fail if it could not be generated.

5.4 Upstream Source Management

This rule allows to fetch, repackage, possibly with files exclusion, the upstream sources. You may also check the source tarball md5sum. It provides a 'get-orig-source' target, as recommended by the Debian Policy (chapter 4.9).

To use this rule, add these lines to your `debian/rules`:

```
include /usr/share/cdbs/1/rules/upstream-tarball.mk

pkgbranch = 0.88

# Optional: you may override the default tarball name (defaults to DEB_SOURCE_PACKAGE)
DEB_UPSTREAM_PACKAGE = $(DEB_SOURCE_PACKAGE:%-$(pkgbranch)=%)

# Compulsory
DEB_UPSTREAM_URL = http://download.sugarlabs.org/sources/sucrose/glucose/$( ↩
    DEB_UPSTREAM_PACKAGE)
DEB_UPSTREAM_TARBALL_EXTENSION = tar.bz2

# Optional: you can request a md5sum check
DEB_UPSTREAM_TARBALL_MD5 = f50a666c4e1f55b8fc7650258da0c539

# Optional: this list of space-separated files and directories will be removed when  ↩
    repackaging
# (Beware directories MUST end with a final '/' !!!)
DEB_UPSTREAM_REPACKAGE_EXCLUDES = .pc/
```

You can then fetch/repackage the source tarball using **fakeroot debian/rules get-orig-source**. If you specified a non-empty `DEB_UPSTREAM_REPACKAGE_EXCLUDES`, you can check if the current tarball is properly up-to-date and do not contain newly excluded patterns using **fakeroot debian/rules fail-source-not-repackaged**.

Chapter 6

Hall of examples

6.1 Java (with upstream-tarball + utils + debhelper)

With this line in `debian/control.in`:

```
Build-Depends: @cdbs@, sharutils
```

`debian/control` management gives in `debian/control`:

```
Build-Depends: cdbs (>= 0.4.70~),
 devscripts (>= 2.10.7~),
 dh-buildinfo,
 debhelper (>= 6),
 ant,
 default-jdk,
 sharutils
```

`debian/rules`:

```
#!/usr/bin/make -f
# -*- mode: makefile; coding: utf-8 -*-
# Copyright © 2003-2010 Jonas Smedegaard <dr@jones.dk>

include /usr/share/cdbs/1/rules/upstream-tarball.mk
include /usr/share/cdbs/1/rules/utils.mk
include /usr/share/cdbs/1/class/ant.mk
include /usr/share/cdbs/1/rules/debhelper.mk

DEB_UPSTREAM_URL = http://prdownloads.sourceforge.net/awstats
DEB_UPSTREAM_TARBALL_BASENAME_MANGLE = s/(-6\.9)\.(\d)/$$1$$2/
DEB_UPSTREAM_TARBALL_MD5 = 26a5b19fa9f395e9e7dafed37b795d7f

DEB_UPSTREAM_REPACKAGE_EXCLUDE = wwwroot/icon/browser/firefox.png

DEB_INSTALL_CHANGELOGS_ALL = docs/awstats_changelog.txt
DEB_INSTALL_DOCS_ALL = README.TXT

awstats_example_scripts = $(filter-out %/awstats_buildstaticpages.pl,$(wildcard tools/*.pl) ↩
    ) $(wildcard debian/examples/*.sh)

DEB_INSTALL_EXAMPLES_awstats = $(awstats_example_scripts) debian/examples/apache.conf  ↩
    wwwroot/cgi-bin/awstats.model.conf wwwroot/cgi-bin/plugins/example/* wwwroot/css wwwroot ↩
    /js tools/xslt
DEB_COMPRESS_EXCLUDE = $(notdir $(awstats_example_scripts)) awstats.ico
```

```
JAVA_HOME = /usr/lib/jvm/default-java
DEB_ANT_BUILDFILE = debian/build.xml
DEB_ANT_BUILD_TARGET = bin-jar
DEB_ANT_ARGS = -Dpackage=awgraphapplet -Dversion=$(DEB_NOEPOCH_VERSION)

# "Binarize" (and cleanup) Debian-shipped non-trademarked Firefox icon
pre-build::
        uudecode -o debian/icons/firefox.png debian/icons/firefox.png.uu
clean::
        rm -f debian/icons/firefox.png

# Adjust for Debian (and cleanup) main config file
common-configure-indep::
        # Use perl rather than a diff here, to make sure all relevant
        # options are checked (upstream defaults have moved around in
        # the past)
# Perl in shell in make requires extra care:
#  * Single-quoting ('...') protects against shell expansion
#  * Double-dollar ($$) expands to plain dollar ($) in make
        perl -wp \
                -e 's,^(LogFile\s*=\s*).*,$$1"/var/log/apache2/access.log",;' \
                -e 's,^(DirData\s*=\s*).*,$$1"/var/lib/awstats",;' \
                -e 's,^(LogFormat\s*=\s*).*,$${1}4,;' \
                -e 's,^(DNSLookup\s*=\s*).*,$${1}1,;' \
                -e 's,^(DirIcons\s*=\s*).*,$$1"/awstats-icon",;' \
                -e 's,^(HostAliases\s*=\s*).*,$$1"localhost 127.0.0.1",;' \
                -e 's,^(DirLang\s*=\s*).*,$$1"/usr/share/awstats/lang",;' \
                -e 's,^#(LoadPlugin\s*=\s*"hashfiles"),$$1,;' \
                -e 's,^#(Include\s*)"",$$1"/etc/awstats/awstats.conf.local",;' \
                -e 's,^(#LoadPlugin\s*=\s*"geoip\s*GEOIP_STANDARD\s*).*,$$1/usr/share/GeoIP ↩
                    /GeoIP.dat",;' \
                -e 's,^(#LoadPlugin\s*=\s*"geoip_region_maxmind\s*GEOIP_STANDARD\s*).*,$$1/ ↩
                    usr/share/GeoIP/GeoIPRegion.dat",;' \
                -e 's,^(#LoadPlugin\s*=\s*"geoip_city_maxmind\s*GEOIP_STANDARD\s*).*,$$1/ ↩
                    usr/share/GeoIP/GeoIPCity.dat",;' \
                -e 's,^(#LoadPlugin\s*=\s*"geoip_isp_maxmind\s*GEOIP_STANDARD\s*).*,$$1/usr ↩
                    /share/GeoIP/GeoIPISP.dat",;' \
                -e 's,^(#LoadPlugin\s*=\s*"geoip_org_maxmind\s*GEOIP_STANDARD\s*).*,$$1/usr ↩
                    /share/GeoIP/GeoIPOrg.dat",;' \
                < wwwroot/cgi-bin/awstats.model.conf > debian/awstats.conf
clean::
        rm -f debian/awstats.conf

# Install (and cleanup) java applet
# TODO: use DEB_DH_INSTALL_ARGS_awstats when some day implemented in CDBS
DEB_DH_LINK_awstats = usr/share/java/awstats/awgraphapplet$(DEB_NOEPOCH_VERSION).jar usr/ ↩
    share/java/awstats/awgraphapplet.jar
binary-install/awstats:: DEB_DH_INSTALL_ARGS = wwwroot/classes/awgraphapplet$( ↩
    DEB_NOEPOCH_VERSION).jar usr/share/java/awstats/
clean::
        rm -f wwwroot/classes/awgraphapplet?*.jar

# Remove badly coded PDF and superfluous GPL license texts
binary-post-install/awstats::
        find $(DEB_DESTDIR) -type f \
                \( -name '*.pdf' -o -name COPYING.TXT -o -name LICENSE.TXT \) \
                -exec rm '{}' +

# Set scripts executable - and unset other files
binary-fixup/awstats::
        find $(DEB_DESTDIR)/usr/share/awstats -type f  -exec chmod -x '{}' +
```

```
        find $(DEB_DESTDIR)/usr/share/awstats/tools $(DEB_DESTDIR)/usr/share/doc/awstats/ ↩
            examples \
                \( -name '*.pl' -o -name '*.sh' \) -exec chmod +x '{}' +
```

6.2 Ruby (with debhelper)

debian/rules:

```
#!/usr/bin/make -f
# -*- mode: makefile; coding: utf-8 -*-

include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/ruby-pkg-tools/1/class/ruby-setup-rb.mk

DEB_RUBY_CONFIG_ARGS += --shebang=never

$(patsubst %,install/%,$(DEB_RUBY_REAL_LIB_PACKAGES)) :: install/% :
        chmod a+x debian/$(cdbs_curpkg)/$(DEB_RUBY_LIBDIR)/feed_tools/vendor/html5/bin/*
        sed -i "s/^#!\/usr\/bin\/env *ruby$$/#!\/usr\/bin\/env ruby$(cdbs_ruby_ver)/"  ↩
            debian/$(cdbs_curpkg)/$(DEB_RUBY_LIBDIR)/feed_tools/vendor/html5/bin/*
```

6.3 GNOME (uses autotools)

6.3.1 `gnome-panel` package

debian/rules:

```
#!/usr/bin/make -f

include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/rules/utils.mk
include /usr/share/cdbs/1/class/gnome.mk
include /usr/share/gnome-pkg-tools/1/rules/uploaders.mk
include /usr/share/gnome-pkg-tools/1/rules/gnome-version.mk
-include /usr/share/gnome-pkg-tools/1/rules/gnome-get-source.mk

LDFLAGS += -Wl,-z,defs -Wl,-O1 -Wl,--as-needed

DEB_CONFIGURE_EXTRA_FLAGS += \
                        --disable-scrollkeeper \
                        --with-in-process-applets=all
ifneq ($(DEB_BUILD_GNU_SYSTEM),gnu)
        DEB_CONFIGURE_EXTRA_FLAGS += --enable-eds
endif

DEB_DH_MAKESHLIBS_ARGS_libpanel-applet2-0 += -V"libpanel-applet2-0 (>= 2.28.0)"
DEB_DH_MAKESHLIBS_ARGS_gnome-panel += --no-act

binary-install/gnome-panel::
        chmod a+x debian/gnome-panel/usr/lib/gnome-panel/*

binary-install/gnome-panel-data::
        find debian/gnome-panel-data/usr/share -type f -exec chmod -R a-x {} \;
        cd debian/gnome-panel-data/usr/share/gconf && \
                mkdir defaults && \
                mv schemas/panel-default-setup.entries \
```

```
                    defaults/05_panel-default-setup.entries

binary-install/libpanel-applet2-doc::
        find debian/libpanel-applet2-doc/usr/share/doc/libpanel-applet2-doc/ -name ".arch- ↩
            ids" -depth -exec rm -rf {} \;
```

6.3.2 gdm3 package

debian/rules:

```
#!/usr/bin/make -f

include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/class/gnome.mk
include /usr/share/gnome-pkg-tools/1/rules/uploaders.mk
include /usr/share/gnome-pkg-tools/1/rules/gnome-version.mk
include /usr/share/gnome-pkg-tools/1/rules/patch-translations.mk
-include /usr/share/gnome-pkg-tools/1/rules/gnome-get-source.mk

GNOME_MODULE := gdm

DEB_CONFIGURE_SCRIPT_ENV += X_PATH="/usr/bin" \
                            X_SERVER_PATH="/usr/bin" \
                            X_SERVER="/usr/bin/Xorg"

ifeq (linux,$(DEB_HOST_ARCH_OS))
    DEB_CONFIGURE_SCRIPT_ENV += X_CONFIG_OPTIONS="-audit 0 -novtswitch"
else
    DEB_CONFIGURE_SCRIPT_ENV += X_CONFIG_OPTIONS="-audit 0"
endif

DEB_CONFIGURE_EXTRA_FLAGS += --disable-scrollkeeper \
                --enable-ipv6=yes \
                --with-at-spi-registryd-directory=/usr/lib/at-spi \
                --with-default-path=/usr/local/bin:/usr/bin:/bin:/usr/games \
                --with-custom-conf=/etc/gdm3/daemon.conf \
                --with-sysconfsubdir=gdm3 \
                --with-working-directory=/var/lib/gdm3 \
                --with-xauth-dir=/var/run/gdm3 \
                --with-pid-file=/var/run/gdm3.pid \
                --with-log-dir=/var/log/gdm3 \
                --with-screenshot-dir=/var/run/gdm3/greeter \
                --with-defaults-conf=/usr/share/gdm/defaults.conf \
                --with-user=Debian-gdm --with-group=Debian-gdm \
                --with-pam-domain=gdm3

DEB_MAKE_EXTRA_ARGS += authdir=/var/lib/gdm3

ifeq (linux,$(DEB_HOST_ARCH_OS))
        DEB_CONFIGURE_EXTRA_FLAGS += --with-selinux
endif


DEB_DH_INSTALLINIT_ARGS := --noscripts

binary-install/gdm3::
        chmod 755 debian/gdm3/usr/share/gdm/gdmXnestWrapper
        mv debian/gdm3/usr/share/applications/gdmsetup.desktop \
                debian/gdm3/usr/share/gdm/applications/
        rmdir debian/gdm3/usr/share/applications
```

```
        mv debian/gdm3/usr/sbin/gdm-binary debian/gdm3/usr/sbin/gdm3
        cd debian/gdm3/usr/sbin && rm -f gdm-restart gdm-stop gdm-safe-restart
        chmod 755 debian/gdm3/etc/gdm3/Xsession
        dh_installpam -pgdm3 --name=gdm3-autologin
        rm -rf debian/gdm3/var/lib/gdm3/.g*
        rm -rf debian/gdm3/var/run
        rm -f debian/gdm3/usr/sbin/gdm
        rm -f debian/gdm3/etc/pam.d/gdm
        rm -f debian/gdm3/etc/pam.d/gdm-autologin
        cd debian/gdm3/usr/share/gdm/greeter-config && \
                mv session-setup.entries 10_upstream.entries

INFILES := $(wildcard debian/*.desktop.in)
OUTFILES := $(INFILES:.desktop.in=.desktop)

%.desktop: %.desktop.in
        intltool-merge -d debian/po-up $< $@

build/gdm3:: $(OUTFILES)

MANPAGES := $(patsubst %.pod,%,$(wildcard debian/*.pod))

common-build-arch:: $(MANPAGES)

clean::
        rm -f $(MANPAGES)
        rm -f $(OUTFILES)

%: %.pod
        pod2man --section=$(shell echo $@ | sed 's/.*\.//') \
                --release="GNOME $(DEB_GNOME_VERSION)" \
                --center="Debian GNU/Linux" \
                $< \
                | sed -e 's/debian:://' >$@
```

6.4 Python (with debhelper)

debian/rules:

```
#!/usr/bin/make -f
DEB_PYTHON_SYSTEM=pysupport
DEB_COMPRESS_EXCLUDE= .js .inv

include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/class/python-distutils.mk

DEB_PYTHON_INSTALL_ARGS_ALL += --install-layout=deb

install/python-cairo-dev::
        python setup.py install_data --install-dir=$(CURDIR)/debian/tmp/usr
        sphinx-build -bhtml doc build/html

build/python-cairo-dbg::
        set -e; \
        for i in $(cdbs_python_build_versions); do \
          python$$i-dbg ./setup.py build; \
        done

install/python-cairo-dbg::
        for i in $(cdbs_python_build_versions); do \
```

```
          python$$i-dbg ./setup.py install --install-layout=deb --root $(CURDIR)/debian/tmp ↩
              ; \
        done

clean::
        -for i in $(cdbs_python_build_versions); do \
          python$$i-dbg ./setup.py clean -a; \
        done
        rm -f $(CURDIR)/pycairo.pc $(CURDIR)/src/config.h
        rm -rf $(CURDIR)/build
```

6.5 Simple Makefile (with dpatch)

debian/rules:

```
#!/usr/bin/make -f

# to re-generate debian/control, invoke
# fakeroot debian/rules debian/control DEB_AUTO_UPDATE_DEBIAN_CONTROL:=yes

# automatic debian/control generation disabled, cdbs bug #311724.

DEB_MAKE_CLEAN_TARGET    := clean
DEB_MAKE_BUILD_TARGET    := standalone
DEB_MAKE_INSTALL_TARGET  := install INSTALL_PREFIX=$(CURDIR)/debian/apg/usr

include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/rules/dpatch.mk
include /usr/share/cdbs/1/class/makefile.mk

cleanbuilddir/apg::
        rm -f build-stamp configure-stamp php.tar.gz

install/apg::
        mv $(CURDIR)/debian/apg/usr/bin/apg $(CURDIR)/debian/apg/usr/lib/apg/apg
        tar --create --gzip --file php.tar.gz --directory $(CURDIR)/php/apgonline/ .
        install -D --mode=0644 php.tar.gz $(CURDIR)/debian/apg/usr/share/doc/apg/php.tar.gz
        rm php.tar.gz
        install -D --mode=0755 $(CURDIR)/debian/apg.wrapper $(CURDIR)/debian/apg/usr/bin/ ↩
            apg
        install -D --mode=0644 $(CURDIR)/debian/apg.conf $(CURDIR)/debian/apg/etc/apg.conf
```

6.6 Perl (with upstream-tarball + utils + debhelper)

debian/rules:

```
#!/usr/bin/make -f

include /usr/share/cdbs/1/rules/upstream-tarball.mk
include /usr/share/cdbs/1/rules/utils.mk
include /usr/share/cdbs/1/class/perl-build.mk
include /usr/share/cdbs/1/rules/debhelper.mk

DEB_UPSTREAM_PACKAGE = NetSDS-Kannel
#DEB_UPSTREAM_URL = http://www.cpan.org/modules/by-module/NetSDS
DEB_UPSTREAM_URL = http://search.cpan.org/CPAN/authors/id/R/RA/RATTLER
DEB_UPSTREAM_TARBALL_MD5 = 71428ec66538cb4384d47d9a29b12230
```

```
# supress regression tests, currently too broken
DEB_PERL_CHECK_TARGET =
```

Chapter 7

Useful tools

7.1 cdbs-edit-patch (provided with CDBS)

This script is intended to help lazy people edit or easily create patches for the simple-patchsys patch system.

Invoke this script with the name of the patch as argument, and you will enter a copy of your work directory in a subshell where you can edit sources. When your work is done and you are satisfied with your changes, just exit the subshell and you will get back to normal world with `debian/patches/patch_name.patch` created or modified accordingly. The script takes care to apply previous patches (ordered patches needed !), current patch if already existing (in case you want to update it), then generate an incremental diff to only get desired modifications. If you want to cancel the patch creation / modification, you only need to exit the subshell with a non-zero value and the diff will not be generated (only cleanups will be done).

Chapter 8

Conclusion

CDBS solves most common problems and is very pleasant to use. More and more DD are using it, not because they are obliged to, but because they tasted and found it could improve their packages and avoid loosing time on designing silly and complicated rules.

CDBS is not perfect, the BTS entry is not clear, but fixing a single bug most of the time fix a problem for plenty of other packages. CDBS is not yet capable of handling very complicated situations (like packages where multiple C/C++ builds with different options and/or patches are required), but this only affects a very small number of packages. These limitations would be solved in CDBS2, which is work in progress (please contact Jeff Bailey jbailey@raspberryginger.com if you want to help).

Using CDBS more widely would improve Debian's overall quality. Don't hesitate trying it, talking to your friends about it, and contributing.

Have a Lot of FUN with CDBS !!! :-)

Thanks

Thanks to Jeff for his patience and for replying my so many questions.

Special thanks to GuiHome for his help to review this documentation.

This document is a DocBook application, checked using xmllint (from libxml2), produced using xsltproc (from libxslt), using the N. Walsh and dblatex XLST stylesheets, and converted with LaTeX tools (latex, mkindex, pdflatex & dvips) / pstotext (with GS).