La documentation de CDBS

Le site hébergeant la doc ayant disparu … vous trouverez ci-dessous l'export de la doc au format HTML depuis les sources.

Voir aussi la doc en ligne archivée sur https://web.archive.org/web/20170331113853/https://cdbs-doc.duckcorp.org/en/cdbs-doc.xhtml qui n'est pas la même que celle qui se trouve ci-dessous !!!!

The Common Debian Build System

Marc (Duck) Dequènes

DuckCorp
<duck@duckcorp.org>

Arnaud (Rtp) Patard

<arnaud.patard@rtp-net.org>

Peter Eisentraut

<petere@debian.org>

Colin Walters

<walters@debian.org>

Table of Contents

List of Figures

Chapter 1. Introduction

Table of Contents

CDBS is essentially a set of makefile fragments that you may include into your `debian/rules` for building Debian packages. Each makefile fragment can have effects in different parts of the package build process.

The motivating factor for CDBS was originally that more and more programs today are created using GNU Autoconf configure scripts and GNU Automake, and as such they are all very similar to configure and build. It was realized that a lot of duplicated code in everyone's `debian/rules` could be factored out. But CDBS isn't only useful for packages that use the GNU autotools. It is a flexible core upon which you can create your own custom build systems.

Why Does CDBS Exist?

The current generally accepted practice for creating new Debian packages is to run **dh_make**, which generates a bunch of files, the most important of which are `debian/control`, `debian/copyright`, and `debian/rules`. The first two are relatively straightforward. But `debian/rules` is not. Debhelper was an enormous step forward in this area, greatly reducing redundant and incomprehensible code from the Debian package creation process. But it doesn't go far enough; the

typical **dh_make** generated `debian/rules` is hundreds of lines, only some of which apply. From experience with helping several people to learn Debian packaging, `debian/rules` was by far the hardest part for them to understand.

Moreover, this generated code will become stale with time, as the Debian policy changes. At some time in the past, the `DEB_BUILD_OPTIONS` variable dropped the `debug` flag in favor of `noopt`. But gradually it was realized that since the code to check this variable was duplicated over hundreds (if not thousands) of **dh_make**-generated source packages (and had possibly been modified), it would likely be years before most packages were updated. And there are many packages that predate `DEB_BUILD_OPTIONS` and don't even use it at all, when they easily could.

The Unix and hacker cultures in general have long looked down upon generated code, and for good reason. It is often hard to edit, and there is almost always no way to regenerate the code but keep your local changes. Instead of generating code (like all the Microsoft tools tend to do), the Unix tradition is to create a metalanguage, a compiler, or some other form of abstraction.

CDBS is that abstraction. It's not the first attempt at abstracting the Debian build process; before Debhelper, many attempts came along and had only marginal success, if any. So now the question you're asking yourself is probably:

What Makes CDBS Better?

First of all, it is not monolithic (as opposed to debstd, for instance). CDBS is quite simply a set of makefile fragments that can be included; if you don't want a particular part, you just don't include the makefile fragment for it.

Second, CDBS does not attempt to supplant Debhelper (which has generally done an excellent job at the `binary` stage of Debian package building). CDBS can optionally use Debhelper to implement various parts of building a Debian package. This is the recommended mode of operation, actually. But some people may find Debhelper doesn't work for them; if that's the case, you just don't include `debhelper.mk`, and you can do the work yourself.

Third, CDBS tries to make the common case easy. If you have a package that uses the GNU autotools and such, you can have a working build system simply by including about two or three makefile fragments. No custom code required at all. Additionally, CDBS has even higher-level makefile fragments; for example, there are `gnome.mk` and `kde.mk` rule files which handle a number of common things required by GNOME and KDE packages.

Finally, CDBS (along with Debhelper) should make it much easier to effect systemwide changes. For example, if we later decide to switch our default `i386` architecture to `i486` (as we probably will), all we have to do is change `autotools.mk`, and the correct `--host` and `--build` will be passed to all `./configure` invocations. Currently *some* packages have the `DEB_HOST_ARCH` boilerplate code in their `debian/rules`, but many don't.

Why Not Just Debhelper?

Some things done in CDBS could just as well go into a **dh_foo** program (for example, some of `autotools.mk`). Likewise, some **dh_foo** programs would probably do better as CDBS makefile

fragments (**dh_pysupport** and **dh_pycentral** comes to mind).

But CDBS' makefile fragment orientation allows it to do things that Debhelper can't, or can't easily do. For example, CDBS automatically generates a ton of makefile rules corresponding to package building. This saves a great deal of redundant code in `debian/rules`.

CDBS automatically creates `build-arch` and `build-indep` targets, and builds architecture-dependent and -independent packages under them. It also can cleanly affect a number of different parts of the build system (e.g., `clean`, `configure`, `build`) by simply including one makefile fragment; doing this as a **dh_foo** would require inserting a call like `dh_foo --clean`, `dh_foo --configure` at each step. And doing things this way wouldn't allow future expansion; you'd have to change your code to say `dh_foo --build` if the foo helper wanted to modify the build process, too.

So CDBS complements Debhelper (or it can; again, CDBS does not require Debhelper).

Summary

In summary, we believe CDBS (typically combined with Debhelper) is the future of Debian packaging. By reducing the complexity in each package, we make sweeping changes much easier. Debian has made several major transitions in the past, and will in the future. It shouldn't be as painful as it has been. Moreover, CDBS makes creating simple packages very easy, as it should be.

CDBS advantages:

- short and readable `debian/rules`
- automates Debhelper and autotools for you so you don't have to bother with these repetitive tasks
- maintainer can focus on real packaging problems because CDBS helps you but does not limit customization
- rules used in CDBS have been well tested
- switching to CDBS is easy
- can be used to generate debian files (like `debian/control` for GNOME Team Uploaders inclusion)
- CDBS is easily extensible
- It |70>< !!!

Chapter 2. First Steps

Table of Contents

The best documentation for CDBS are the makefile fragments under `/usr/share/cdbs/1/`. The following will tell you how to get started with CDBS and explain what is possible, but since you can pretty much override or customize anything in CDBS, you will sooner or later want to look at the code itself.

Switching the Package to CDBS

Switching to CDBS is easy. A simple `debian/rules` for an autotools-using C or 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
```

Yes, really. This is sufficient, and it handles autotools management, like updating `config.guess` and `config.sub`, cleaning up temporary files after the build, and running all common debhelper commands.

Create your *package*`.install`, *package*`.info`, etc. as you usually do with **dh_*** commands, and CDBS will call them if necessary, autodetecting a lot of things. In case of missing Debhelper compatibility information, CDBS will create a `debian/compat` file with compatibility level 5.

Incidentally, you should usually include `debhelper.mk` first, before other files. This will turn on optional Debhelper-using parts of other rules files, if any, which is usually what you want.

Naturally, if you switch a package to use CDBS, you must add a build dependency on `cdbs` to your package.

Versioning note

CDBS might change incompatibly in the future, and to allow for this, all the rules and classes are in a version-specific subdirectory. That's the reason for the 1 in `/usr/share/cdbs/1`.

Warning

CDBS does not work if the name of the current directory contains spaces or some other special characters such as quotation marks. This situation is very hard to handle in **make**, so it is best to stick to “sane” directory names.

Basic Settings and Available Variables

Every CDBS-using `debian/rules` should eventually include `/usr/share/cdbs/1/rules/buildcore.mk`. (It might be included automatically via dependencies, as we will see later.) This makefile fragment sets up all of the core default makefile structure and variables, but doesn't actually *do* anything on its own.

You can use the `buildcore.mk` rules to hook in your own build system to actually implement each stage of compiling, installing, and building `.deb`s if you wish, but typically you will use the rules and classes that CDBS has prepared for you.

You can change common build parameters this way:

```
# where sources are
DEB_SRCDIR = src

# in which directory to build
DEB_BUILDDIR = $(DEB_SRCDIR)/build
```

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

Note that the variables should be set *after* the rule fragments are included. This is necessary for them to have any effect. There are a few exceptions to this; but generally variables should be set after rule fragments are included.

Some of the variables you can use in `debian/rules`:

`DEB_SOURCE_PACKAGE`

: name of the source package

`DEB_VERSION`

: full Debian version

`DEB_NOEPOCH_VERSION`

Debian version without epoch

`DEB_ISNATIVE`

nonemtpy if the package is native

`DEB_ALL_PACKAGES`

list of all binary packages

`DEB_INDEP_PACKAGES`

list of architecture-independent binary packages

`DEB_ARCH_PACKAGES`

list of architecture-dependent binary packages

`DEB_PACKAGES`

list of normal (non-udeb) binary packages

`DEB_UDEB_PACKAGES`

list of udeb binary packages, if any

`DEB_ARCH`

the old Debian architecture name (deprecated, only use to provide backward compatibility; see **dpkg-architecture** man page for more information)

`DEB_HOST_ARCH_CPU`

the CPU part of the Debian architecture (e.g., `powerpc`)

`DEB_HOST_ARCH_OS`

the operating system part of the Debian architecture (e.g., `linux`)

`DEB_DESTDIR`

The directory in which to install the package. This is automatically set to `$(CURDIR)/debian/`*packagename* if there is one package and `$(CURDIR)/debian/tmp` if there are multiple packages. In the latter case you would usually use `.install` files to install the package files into the final directories. In some exceptional cases, you may need to set `DEB_DESTDIR` yourself. One case is when some packages are only built on some architectures, which could make CDBS set the variable inconsistently across architectures, which in turn would create problems with the `.install` files.

Custom Rules

Suppose that your package generates extra cruft as a side effect of the build process that's not taken care of by the upstream `clean` rule and ends up bloating your diff. To handle this (until upstream fixes it), you can simply add stuff to the `clean` rule as follows:

```
clean::
    rm -f foo/blah.o po/.intltool-merge-cache
```

Almost all of the current rules are *double-colon* rules (see the *GNU Make Manual*). This means you can simply add to them without overriding the default.

You can also add dependencies to the rules. For example, suppose you have a multiple-binary package that builds both a program and a shared library, and the program depends on the shared library. To tell CDBS to build the shared library before the program, just do something like:

```
binary/program:: binary/libfoo
```

However, this must come *before* you include `buildcore.mk`. This is due to the way make works.

Targets of the form *`something/package`* exist for many stages of the package build process and allow you to hook in additional commands. Suppose you want custom rules for the source package `foo`, creating binary packages `foo` (architecture-depependent) and `foo-data` (architecture-independent). You simply need to add some lines like the following 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 (although such a `configure` script would normally also be called in that very package configuration phase).

To add post-build actions:

```
build/foo::
    $(SHELL) 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 {} \;
    strip --remove-section=.comment --remove-section=.note --strip-unneeded
\
        debian/foo/usr/lib/foo/totoz.so

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 (usually not very useful):

```
binary/foo::
    @echo 'Package foo successfully built.'
```

To add pre-clean actions:

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

Now, let's suppose your package is a little bit strange (e.g. Perl); perhaps it has a **Configure** script that isn't made by Autoconf; this script might instead expect the user to interactively configure the program. In that case, you can just implement the `common-configure` rule, by adding something like the following to your `debian/rules`:

```
common-configure::
    ./Configure --blah --blargh < debian/answers
```

Note that if you do this, you can't include `autotools.mk`, since then you'll get *two* implementations of common-configure, which is sure to fail. It would be nice to be able to partially override rule fragments, but that is a tricky problem.

Figure 2.1, “Buildcore targets” gives an overview of the targets provided by `buildcore.mk` where you can hook in custom rules. The bold ellipses are the targets required by the Debian policy. For illustration, the diamond-shaped nodes show how a typical autotools-using build process would be hooked into these rules. These are not actually provided by `buildcore.mk` of course.

Figure 2.1. Buildcore targets

Rules that add commands should normally be written *after* including any CDBS rule fragments, unless you know exactly what you're doing. The reason for this is as follows. The commands for double-colon rules are accumulated in the order in which they appear in the makefile. That is, writing

```
target::
    foo

target::
    bar
```

will have approximately the same effect as

```
target:
    foo
    bar
```

Now if you have, for example, an autotools-using package and wrote something like

```
build/myprog::
    $(MAKE) extrastuff

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

(more on the autotools class below), this would end up running `$(MAKE) extrastuff` before `autotools.mk` has a chance to run **configure**, so there will probably not be any instantiated makefile yet and the build will fail.

Again, the recommended practice is to include all the CDBS rule fragments first on your `debian/rules` and put variable assignments and extra rules afterwards, unless an exception is explicitly pointed out in this manual.

Common Build Options

`CFLAGS` and `CXXFLAGS` are set to `-g -Wall -O2` by default.

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

Debhelper Support

An important piece of the puzzle after configuring and building the software is to actually build `.debs` from there. You could implement this step yourself if you wished, but most people will want to take advantage of Debhelper to do it mostly automatically. To do this, simply add a line like

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

in `debian/rules`.

CDBS debhelper rules handle the following **dh_*** commands for each binary package automatically:

- dh_builddeb
- dh_clean
- dh_compress
- dh_fixperms
- dh_gencontrol
- dh_install
- dh_installcatalogs
- dh_installchangelogs
- dh_installcron
- dh_installdeb
- dh_installdebconf
- dh_installdirs
- dh_installdocs
- dh_installemacsen
- dh_installexamples
- dh_installinfo
- dh_installinit

- dh_installlogcheck
- dh_installlogrotate
- dh_installman
- dh_installmenu
- dh_installmime
- dh_installpam
- dh_installudev
- dh_link
- dh_lintian
- dh_makeshlibs
- dh_md5sums
- dh_perl
- dh_prep
- dh_shlibdeps
- dh_strip

Of course, these are called in the correct order, not in the one shown above. Other debhelper commands can be handled in specific classes or may be called in custom rules.

If you use `debhelper.mk`, you must add a build dependency on `debhelper`. If you use Debhelper compatibility level 5, then the dependency should be (at least) `debhelper (>= 5)`, if you use version 4 then (at least) `debhelper (>= 4.2.0)`.

Debhelper Parameters

The following variables allow Debhelper call customization while leaving the other calls to be handled without writing any rule. Some of them apply on all binary packages, for instance `DEB_INSTALL_DOCS_ALL`, and some apply only to a specific package, for instance `DEB_SHLIBDEPS_LIBRARY_`*package* (where *package* is the name of a binary package). Read the comments in `/usr/share/cdbs/1/rules/debhelper.mk` for a complete listing. Some non-exhaustive examples follow.

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 like `ProjectChanges.txt.gz`:

```
DEB_INSTALL_CHANGELOGS_ALL = ProjectChanges.txt
```

(CDBS automatically recognizes a fair number of possible changelog names, but not that one.)

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

```
DEB_COMPRESS_EXCLUDE = .py
```

Perl-specific debhelper options (The **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
```

Debhelper Custom Build Rules

CDBS debhelper rules also add more adequate build rules. For example, to add post deb preparation (including debhelper stuff) actions:

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

Several other rules exists, for rarer cases:

- `binary-strip/foo` (called after stripping)
- `binary-fixup/foo` (called after gzipping and fixing permissions)
- `binary-predeb` (called just before creating the `.deb`)

Debug Package Support

A debug package is a binary package named *package*`-dbg` that contains the debugging symbols for the binaries (programs, libraries, etc.) in other packages, typically all other binary packages built from the same source package. Debhelper facilitates the creation of these debug packages by the `--dbg-package` option in the command **dh_strip**. CDBS has support for creating debug packages, if Debhelper level 5 compatibility is used.

CDBS will automatically call **dh_strip** with the right options if exactly one debug package is mentioned in `debian/control` and so that the debugging symbols of all other binary packages are included in that debug package. This takes care of the most common situation.

To control more finely which debug symbols go where, in particular if you want to build more than one debug package, there are variables DEB_DBG_PACKAGE_*package* that specify the debug package target for each individual binary package. An example usage would be:

```
DEB_DBG_PACKAGE_libfoo4 = libfoo-dbg
DEB_DBG_PACKAGE_foo-bin = foo-bin-dbg
```

If exactly one debug package is defined, setting any variable DEB_DBG_PACKAGE_*package* disables the behavior of putting all debug symbols in that package.

If there is more than one debug package defined and each debug package is named `foo-dbg` such that there is a package called `foo`, then the assignments `DEB_DBG_PACKAGE_foo = foo-dbg` are done automatically. Again, this only happens if all debug packages can be assigned this way. Of course, all these assignments can be overridden if you find that this behavior doesn't quite work for you.

Note

If a source package builds a single binary package, and then a debug package is added, this changes the automatic assignment of DEB_DESTDIR to `$(CURDIR)/debian/tmp`, as described in the section called “Basic Settings and Available Variables”, which will likely invalidate the installation rules and leave you with a nearly-empty package. To work around this behavior, set DEB_DESTDIR manually in `debian/rules` as

```
DEB_DESTDIR = $(CURDIR)/debian/packagename
```

Alternatively, write a *packagename*`.install` file listing

```
debian/tmp/*
```

or whatever subset you need.

Chapter 3. Classes

Table of Contents

CDBS provides *classes* which contain makefile rules and variables implementing compilation, installation, and building of Debian packages. There are a number of classes covering different types of ways a software is built. Classes tend to be declarative; they say your package has particular properties. Suppose for instance that your package uses a regular makefile to compile, and has the normal **make** and **make install** targets. In that case you would use the “makefile” class, and you can say:

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

This gives you all the code to run **make** automatically. It basically works by adding code to the `common-build-arch`, `common-build-indep`, `common-install-arch`, and `common-install-indep` targets inside `buildcore.mk`. It might be instructive to look at `makefile.mk` now.

Some classes actually include another class, or “inherit” if you like. For example, the autotools class inherits the makefile class because much of the build process is the same between them, only the

configuration stage is different. The effect is that all the variables provided by the inherited class are available in the inheriting class as well. Figure 3.1, “Dependencies between the CDBS class and rule files” shows the relationship between the classes and other rule sets provided by CDBS.

Figure 3.1. Dependencies between the CDBS class and rule files

The rest of this chapter explains all the classes supported by CDBS.

The Makefile Class

The makefile class is for the packages who only have a makefile to build the program. (If the package uses Autoconf, use the autotools class instead.) You only need to have four rules in the makefile:

- one for cleaning the build directory (e.g., `clean`)
- one for building the software (e.g. `all`)
- one for checking if the software is working properly (e.g. `check`)
- one for installing the software (e.g. `install`)

The installation and check rules are optional, but it always helps a lot when you've got them.

The first step 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   = clean
DEB_MAKE_BUILD_TARGET   = all
DEB_MAKE_INSTALL_TARGET = install DESTDIR=$(CURDIR)/debian/tmp/
# no check for this software
DEB_MAKE_CHECK_TARGET   =

# example when changing environment variables is necessary
DEB_MAKE_ENVVARS        = CFLAGS="-pwet"
```

`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 makefile.

Build Problems

Sometimes, when using the makefile class (or a derived one), a build fails because of missing include files or something like that. Often this is caused by the fact that CDBS passes `CFLAGS` (and `CPPFLAGS`) along with the make invocation. A sane build system allows this: `CFLAGS` are for the user to customize. Setting `CFLAGS` shouldn't override other internal flags used in the package, such as `-I`. If fixing the upstream source is too difficult, however, you may do this:

```
DEB_MAKE_INVOKE = $(DEB_MAKE_ENVVARS) make -C $(DEB_BUILDDIR)
```

That will avoid passing `CFLAGS`. But note that this breaks the automatic implementation of `DEB_BUILD_OPTIONS`.

The Makefile Class and Single vs. Multiple Binary Packages

If you have a single binary package, the default `common-install` implementation in `makefile.mk` tries to use the upstream `Makefile` to install everything into `debian/packagename`, so it will all appear in the binary package. If you're using `debhelper.mk` to remove files or move them around, just override the `binary-post-install/`*`package`* target:

```
binary-post-install/mypackage::
    mv debian/mypackage/usr/sbin/myprogram
debian/mypackage/usr/bin/myprogram
    rm debian/mypackage/usr/share/doc/mypackage/INSTALL
```

If you have a multiple-binary package, `makefile.mk` (by default) uses the upstream `Makefile` to install everything in `debian/tmp`. After this, the recommended method is to use `debhelper.mk` (which uses **dh_install**) to copy these files into the appropriate package. To do this, just create *`package`*`.install` files; see the **dh_install** man page.

The Autotools Class

The autotools class is for software that uses GNU Autoconf and possibly Automake and Libtool. The class will take care of details such as updating the `config.{sub,guess}` files, running **configure** with the standard arguments, etc. The autotools class actually builds upon the makefile class.

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

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

Suppose you need to pass some additional options to **configure**. The `autotools.mk` file includes a number of variables that you can override for that purpose, like this:

```
DEB_CONFIGURE_EXTRA_FLAGS = --enable-ipv6 --with-foo
```

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

```
DEB_CONFIGURE_NORMAL_ARGS = --program-dir=/usr
```

Note that `DEB_CONFIGURE_EXTRA_FLAGS` will still be appended.

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

```
DEB_CONFIGURE_SCRIPT_ENV += LDFLAGS=" -Wl,-z,defs -Wl,-O1"
```

> **Tip**
>
> Prefer the use of += over = lest you override other environment variables like CC or CXX defined in the CDBS default.

CDBS will automatically update `config.sub`, `config.guess`, and `config.rpath` before the build and restore the old ones at the clean stage (even if using the tarball system). If needed, and if `debian/control` management is activated, `autotools-dev` and `gnulib`, respectively, will then be automatically added to the build dependencies (needed to obtain updated versions of the files). Otherwise, you should add these packages, as appropriate, to the build dependencies yourself. If you fail to do so, these updates will not execute unless the required packages are already installed by coincidence. (Lintian is likely to complain if you forget.) If the program does not use the top source directory to store autoconf files, you can teach CDBS where they are to be found:

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

CDBS automatically cleans autotools files generated during the build (`config.cache`, `config.log`, and `config.status`).

CDBS can be asked to update Autoconf, Automake, and Libtool generated 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_AUTOCONF to the version of Autoconf to use; e.g., `2.61`
- DEB_AUTO_UPDATE_AUTOHEADER to the version of **autoheader** to use; e.g., `2.61`
- DEB_AUTO_UPDATE_AUTOMAKE to the version of Automake to use; e.g., `1.10`. To pass extra arguments to **automake**, such as `--add-missing --copy`, put them into the variable DEB_AUTOMAKE_ARGS.
- DEB_AUTO_UPDATE_ACLOCAL to the version of **aclocal** to use; e.g., `1.10`. In the current version of CDBS, DEB_AUTO_UPDATE_AUTOMAKE implies DEB_AUTO_UPDATE_ACLOCAL. This behavior will eventually be discontinued; so if you meant `aclocal.m4` to be regenerated, please use DEB_AUTO_UPDATE_ACLOCAL.
- DEB_AUTO_UPDATE_LIBTOOL to `pre` to run **libtoolize** before the build, or to `post` to copy the system-supplied **libtool** program into the build tree after the configure run

(Corresponding build dependencies will automatically be added.)

The following make parameters can also 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 unexistant install rule for make
DEB_MAKE_INSTALL_TARGET =

# example to activate check rule
DEB_MAKE_CHECK_TARGET = check
```

DEB_BUILD_OPTIONS is checked for the following options:

`noopt`

use `-O0` instead of `-O2`

`nocheck`

skip the check rule

The CMake Class

CMake is a cross-platform build tool. On Unix-like systems it typically generates makefiles, which are then run through make normally. To use the CMake class, add this include to your `debian/rules`:

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

The class takes care of the call to **cmake** and the subsequent calls to **make**, with all the necessary options to honor `DEB_BUILD_OPTIONS`, for example. To that end, the CMake class builds upon the makefile class.

CMake is designed to always use separate source and build directories. Therefore, the CMake class by default builds the project in a separate build directory named like `obj-`*`platform`* under the top-level source directory.

`DEB_BUILD_OPTIONS` is checked for the following options:

`noopt`

use `-O0` instead of `-O2`

The Perl Class

The Perl class can manage Perl module packages using MakeMaker. To use this class, add this line to your `debian/rules`:

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

The installation directory defaults to *`first_pkg`*`/usr` where *`first_pkg`* is the first package in `debian/control`.

You can customize build options like this:

```
# change MakeMaker defaults (hardly ever useful)
DEB_MAKE_BUILD_TARGET = build-all
DEB_MAKE_CLEAN_TARGET = realclean
DEB_MAKE_CHECK_TARGET =
DEB_MAKE_INSTALL_TARGET = install PREFIX=debian/stuff
```

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

Common makefile or general options can still be overridden: DEB_MAKE_ENVVARS, DEB_BUILDDIR (must match DEB_SRCDIR for Perl).

Have a look at the Perl-specific debhelper options described above.

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

The Python Class

The Python class can manage Python module packages using Distutils. To use this class, add this line to your debian/rules:

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

Optionally, this class can take care of using **dh_pycentral** or **dh_pycentral** as needed, if the CDBS debhelper rules are also included.

Most Python packages are architecture-independent and then don't need to be built for multiple Python versions; your package should then be called python-*foo* and CDBS will automatically use the current Python version in Debian to build it. If your package contains a compiled part or a binding to an external library, then you will have packages named python2.3-*foo*, python2.4-*foo*, and so on, depending on ${python:Depends} (and perhaps other packages); then CDBS will automatically build each package with the corresponding Python version. In this case, don't forget to add a python-*foo* convenience dummy package depending on the current Python version in Debian.

You can customize build options like this:

```
# 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

# specific additional install options for binary package 'foo'
```

```
# ('--root' option is always set)
DEB_PYTHON_INSTALL_ARGS_foo = --root=debian/foo-install-dir/
```

The GNOME Class

The GNOME class is obviously for building GNOME software. It inherits the autotools class, so everything that was said there also applies to the GNOME class.

The GNOME class can call the following debhelper scripts automatically:

- dh_desktop
- dh_gconf
- dh_icons
- dh_scrollkeeper

Moreover it adds some more clean rules:

- to remove **intltool** generated files
- to remove **scrollkeeper** generated files

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

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

The GNOME class adds a make environement 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 disables schema registration in `make install`. This procedure is deferred until **gconftool-2** is called in `debian/postinst` to register them, and in `debian/prerm` to unregister them. The **dh_gconf** command is able to add the right rules automatically for you.

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

The Debian GNOME Team Class

If you are part of the GNOME team or have the team as uploaders, and you feel bored maintaining the list of developers, the GNOME Team class is made for you.

To use this class, 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 (`fakeroot debian/rules clean`) to regenerate the `debian/control` file, which will replace the `@GNOME_TEAM@` tag with the list of developers automatically.

Warning

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

```
# deactivate 'debian/control' file management
#DEB_AUTO_UPDATE_DEBIAN_CONTROL = yes

# ...
# includes and other stuff
# ...

clean::
    sed -i "s/@cdbs@/$(CDBS_BUILD_DEPENDS)/g" debian/control
    # other clean stuff
```

The KDE Class

The support for building KDE-related packages for KDE version 4 and higher using CDBS is included in the package `pkg-kde-tools`. To use the KDE class, add this line to your `debian/rules` file:

```
include /usr/share/pkg-kde-tools/makefiles/1/cdbs/kde.mk
```

and add `pkg-kde-tools` to the build dependencies. The KDE class inherits the cmake class, so everything that was said there also applies here.

The KDE class provides a plethora of options to the **cmake** call that have been chosen so that the resulting packages integrate properly with Debian.

The following files are excluded from compression:

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

Obsolescent: KDE 3 Support

The class named kde.mk included in CDBS is for building KDE-3-based packages. It will eventually be removed from CDBS.

To use it, add this line to debian/rules:

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

This KDE 3 class inherits the autotools class, so everything that was said there also applies here.

CDBS automatically exports the following variables with the right values:

- 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 class handles configure options specific to KDE (not forgetting to disable rpath and activating xinerama), set the correct autotools directory, and launch make rules adequately.

DEB_BUILD_OPTIONS is checked for the following options:

noopt

disable KDE final mode

`nostrip`

enable KDE debug mode and disable KDE final mode

`debug`

enable full KDE debug mode

The Qmake Class

Qmake is a build tool for software written for the Qt toolkit library. To use the Qmake class, add this include to your `debian/rules`:

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

The class takes care of the call to **qmake** and the subsequent calls to **make**, with all the necessary options to honor `DEB_BUILD_OPTIONS`, for example. To that end, the Qmake class builds upon the makefile class.

The Qmake class will call `make install`, but many Qmake projects are not set up to have a functioning install target, in which case the installation of the package components has to be handled manually.

`DEB_BUILD_OPTIONS` is checked for the following options:

`noopt`

use `-O0` instead of `-O2`

`nostrip`

pass the `nostrip` option to **qmake** though the `CONFIG` variable

The Ant Class

Ant is a build tool for software written in the Java programming language. To use the Ant class, add this include to your `debian/rules`:

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

Additionally, you need to set the variable `JAVA_HOME` to the home directory of the Java Runtime

Environment (JRE) or Java Development Kit (JDK). You can either set JAVA_HOME directly or set JAVA_HOME_DIRS to multiple possible home directories. The first directory from this list that provides a **java** command is used for JAVA_HOME. For Ant-using packages in the Debian main archive, you would typically use either

```
JAVA_HOME = /usr/lib/jvm/default-java
```

which requires a build dependency on default-jdk, or

```
JAVA_HOME = /usr/lib/jvm/java-gcj
```

which requires a build dependency on java-gcj-compat-dev. Setting the Java home is required; there is no default.

You can also override JAVACMD in case you don't want to use the default *JAVA_HOME*/bin/java.

You may add additional JARs to the build like in the following example:

```
DEB_JARS = xerces /usr/lib/java-bonus/ldap-connector.jar
```

The .jar extension may be omitted. The file name must be absolute or relative to /usr/share/java. ant.jar, ant-launcher.jar, and $(JAVA_HOME)/lib/tools.jar are automatically added if they exist.

To use a specific Java compiler, override the variable DEB_ANT_COMPILER, for example

```
DEB_ANT_COMPILER = jikes
```

If your package does not put the file build.xml in the package root directory, where Ant would find it by default, you can point CDBS to the right place like this:

```
DEB_ANT_BUILDFILE = build/build.xml
```

Finally, you need to set the targets to invoke for building, installing, testing and cleaning up. Unless overridden, building uses the default target from build.xml, installing and testing is only called if the corresponding variable is set, cleaning uses the clean target. You can also specify multiple targets for each step. To override these rules, or run the install or check rules, set the following variables to your needs:

```
DEB_ANT_BUILD_TARGET = makeitrule
DEB_ANT_CHECK_TARGET = check
DEB_ANT_INSTALL_TARGET = install-all
DEB_ANT_CLEAN_TARGET = super-clean
```

Ant called by CDBS will read a property file, by default at debian/ant.properties if it exists. There you may define additional properties that are referenced from build.xml so that you don't

have to modify upstream's `build.xml`. Please note that command-line arguments in `ANT_ARGS` (see below) override the settings in `build.xml` and the property file. The use a different property file, set the variable `DEB_ANT_PROPERTYFILE`.

You can provide additionnal JVM arguments using the variable `ANT_OPTS`. You can moreover provide additional Ant command line arguments using `ANT_ARGS` (global) or `ANT_ARGS_`*package*, thus overriding the settings in `build.xml` and the property file.

`DEB_BUILD_OPTIONS` is checked for the following options:

`nocheck`

skip the check target

`noopt`

set Ant option `compile.optimize` to false

See the section called “Ant Example” for a complete example that uses this Ant class. You can also get some more information on Ant at the Ant Apache web site.

The HBuild Class

HBuild is the Haskell mini-distutils. CDBS can take care of `-hugs` and `-ghc` packages: invoke `Setup.lhs` properly for the clean and install part.

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

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

You should be able to fetch some more information on Haskell distutils in this thread.

The Waf Class

Waf is a generic and flexible build system based on python. CDBS takes care of calling the configure, build and install steps with correct `PREFIX` and `DESTDIR`.

To use this class, you must include this line to your `debian/rules` file:

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

If you need a more advanced control upon build configuration, you can pass options to the configure step using the built-in variable `DEB_WAF_CONFIGURE_OPTIONS`. You can pass options to each WAF

invocation using the DEB_WAF_OPTIONS variable. You can as well append variables to the environment WAF will run in with the DEB_WAF_ENV variable. Note : those three vars supports global, default and per-package scope. This means you must add the scope suffix to use them. Full debian/rules usage example:

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

include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/class/waf.mk

DEB_WAF_CONFIGURE_OPTIONS_DEFAULT = --enable-this-option --disable-this-one
DEB_WAF_OPTIONS_DEFAULT = --verbose
DEB_WAF_ENV_DEFAULT = ENVVAR=value
```

Warning

Waf design is based on a self-extracting executable within the source tree. Since waf is executed blindly during build process, it implies potential security risk if malware is inserted inside waf. To reduce this risk, CDBS includes a SHA1 checksum of the waf file before the build process. This is done with checking the file debian/waf.sha1sum. The packager has charge to create and maintain this file along the package evolution. debian/waf.sha1sum can be created with following command:

```
sha1sum ./waf > debian/waf.sha1sum
```

However, if the packager trusts the waf file, this checksum can be skipped by assigning a non-zero value to DEB_WAF_SKIP_CHECKSUM.

```
DEB_WAF_SKIP_CHECKSUM = 1
```

More information can be found about WAF in a general way on the project page.

Chapter 4. Common Tasks

Table of Contents

CDBS also supports other tasks that regularly occur during the course of building Debian packages.

Patching Sources Using the Simple Patch System

Suppose you'd like to keep separated patches, instead of having them all in your `.diff.gz`. CDBS lets you hook in arbitrary patch systems, but (as with the rest of CDBS), it has its own default implementation, called `simple-patchsys.mk`. To use it, just add

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

to your `debian/rules`. Now, you can drop patch files into the `debian/patches` directory, and they will be automatically applied and unapplied. Files should be named so as to reflect in which order they have to be applied, and must end in a `.patch` or `.diff` suffix. The simple patchsys rules will then take care of patching before configure and unpatching after clean. It is possible to use patch level 0 to 3, and CDBS will try them and use the right level automatically. The system can handle compressed patches with an additionnal `.gz` or `.bz2` suffix as well as uuencoded patches with a `.uu` suffix.

You can customize the directories where patches are searched and the suffix like this:

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

The defaults are: `.diff`, `.diff.gz`, `.diff.bz2`, `.diff.uu`, `.patch`, `.patch.gz`, `.patch.bz2`, `.patch.uu`.

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

When using the simple patch system, a build dependency on `patchutils` should be added to the package.

The script **cdbs-edit-patch** is intended to help lazy people edit or create patches easily. Invoke this script with the name of the patch as argument, and you will enter a copy of your working directory in a subshell where you can edit the 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!), the 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 or modification, you only need to exit the subshell with a nonzero value and the diff will not be generated (only cleanups will be done).

Patching Sources Using Dpatch

Like the simple patch system detailed previously, the Dpatch patch system allows you to seperate your changes to the upstream tarball into multiple seperate patches instead of a monolithic `diff.gz`.

This is a wrapper to the tools contained in the dpatch Debian package, and it's named dpatch.mk. To use it, add

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

to your debian/rules. If you use autotools.mk, be sure to include dpatch.mk *after* autotools.mk. Additionally, remember to add dpatch to your build dependencies.

Now you can use Dpatch as usual and CDBS will call it automatically. For a more complete treatment of Dpatch files, their format, and their application, please read the documentation included in the dpatch package, notably /usr/share/doc/dpatch/README.gz and the **dpatch** man page.

Patching Sources Using Quilt

Quilt is yet another patch management system. CDBS itself does not actually contain any Quilt support, but the Quilt package contains CDBS integration, so there is really no difference from the perspective of the user. To use Quilt with CDBS, add

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

to your debian/rules and add quilt to the build dependencies. Read the documentation in the Quilt package for more information.

Standard Patch System Targets

The most popular patch systems in Debian, the CDBS Simple Patch System, DPatch, and Quilt, support the following uniform make targets that you can use directly to apply and unapply the patches. This could be useful during package development. Of course, the patches are automatically applied or unapplied as necessary when a full package build is performed.

patch

: to apply the patches

unpatch

: to unapply the patches

Tarball-Inside-a-Tarball Build System

Some Debian packagers may be familiar with DBS, where you include a tarball of the upstream source inside the Debian source package itself. This has some advantages and some disadvantages, but CDBS supports it anyhow. To use the CDBS tarball system, just add this line to your debian/rules,

and specify the name of the top directory of the extracted tarball:

```
DEB_TAR_SRCDIR = foosoft

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

Note that `tarball.mk` must be *first* in the list of included rules.

CDBS will recognize tarballs with the following extensions: `.tar`, `.tgz`, `.tar.gz`, `.tar.bz`, `.tar.bz2`, `.zip`. The tarball location is autodetected if it is in the top source directory, or can be specified:

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

CDBS will handle automatic extraction and cleanups, automatically set `DEB_SRCDIR` and `DEB_BUILDDIR` for you, and take care of proper integration with other CDBS parts (such as the autotools class).

Note that a build dependency on `bzip2` or `unzip` may be in order if that is the format of the tarball. The `gzip` package is essential, so no build dependency is necessary for it.

`debian/control` Management

> **Warning**
>
> This feature is considered broken and packages using it are not allowed into the Debian archive.

The `debian/control` management feature allows:

- CDBS to manage some build dependencies automatically
- use of shell commands embedded in `debian/control`
- use of CPU and system criteria to specify architecture (experimental)

Build dependencies are introduced by the use of certain CDBS features or autodetected.

Embedded shell commands allows including hacks like:

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

CPU and system criteria implement 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.)

Procedure 4.1. `debian/control` Management

1. Rename `debian/control` to `debian/control.in`.

2. Replace `cdbs` and `debhelper` build dependencies 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. Add the following line to `debian/rules`, before *any* include:

   ```
   DEB_AUTO_UPDATE_DEBIAN_CONTROL = yes
   ```

4. Then do a **debian/rules clean** run to (re)generate `debian/control`.

Chapter 5. Hall of Examples

Table of Contents

Makefile + Dpatch example
Perl Examples
Ant Example

There are as of this writing more than a thousand packages in the Debian archive that use CDBS, so there is a rich source of examples. Nonetheless, to complete this manual, here are a few representative examples of real packages using CDBS so you get an idea of how to put these things together.

GNOME + Autotools + Simple Patch System Example

This example is from the package `gnome-panel`.

`debian/control.in`:

```
Source: gnome-panel
Section: gnome
Priority: optional
Maintainer: Guilherme de S. Pastore <guilherme.pastore@terra.com.br>
Uploaders: Sebastien Bacher <seb128@debian.org>, Arnaud Patard
<arnaud.patard@rtp-net.org>, @GNOME_TEAM@
Standards-Version: 3.6.2.1
Build-Depends: @cdbs@, liborbit2-dev (>= 1:2.12.1-1), intltool, gnome-pkg-
tools, libgtk2.0-dev (>= 2.7.1), libglade2-dev (>= 1:2.5.1), libwnck-dev (>=
2.11.91-1), scrollkeeper (>= 0.3.14-9.1), libgnome-desktop-dev (>= 2.11.1),
libpng3-dev, libbonobo2-dev (>= 2.8.1-2), libxmu-dev, libedata-cal1.2-dev
(>= 1.2.1-1) [!hurd-i386], libgnome-menu-dev (>= 2.11.1-1), libgnomevfs2-dev
(>= 2.10.0-1), libnspr-dev, libxres-dev, sharutils, gnome-doc-utils,
libedataserverui1.2-dev (>= 1.3.0)

Package: gnome-panel
Architecture: any
Depends: ${shlibs:Depends}, ${misc:Depends}, gnome-panel-data (= ${Source-
Version}), gnome-desktop-data (>= 2.10.0-1), gnome
-control-center (>= 1:2.8.2-3), gnome-menus (>= 2.11.1-1), gnome-about (>=
2.10.0-1)
Recommends: gnome-applets (>= 2.12.1-1), gnome-session, menu-xdg (>= 0.2)
Suggests: yelp, gnome2-user-guide, gnome-terminal, gnome-system-tools,
nautilus
Description: launcher and docking facility for GNOME 2
...
```

`debian/rules`:

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

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

include /usr/share/cdbs/1/rules/debhelper.mk
# Including this file gets us a simple patch system.  You can just
# drop patches in debian/patches, and they will be automatically
# applied and unapplied.
include /usr/share/cdbs/1/rules/simple-patchsys.mk
# Including this gives us a number of rules typical to a GNOME
# program, including setting GCONF_DISABLE_MAKEFILE_SCHEMA_INSTALL=1.
# Note that this class inherits from autotools.mk and docbookxml.mk,
# so you don't need to include those too.
include /usr/share/cdbs/1/class/gnome.mk

DEB_CONFIGURE_SCRIPT_ENV += LDFLAGS="-Wl,-z,defs -Wl,-O1 -Wl,--as-needed"
DEB_CONFIGURE_EXTRA_FLAGS := --disable-scrollkeeper
ifneq ($(DEB_BUILD_GNU_SYSTEM),gnu)
    DEB_CONFIGURE_EXTRA_FLAGS += --enable-eds
endif

# debug lib
DEB_DH_STRIP_ARGS := --dbg-package=libpanel-applet-2

# tight versioning
DEB_NOREVISION_VERSION := $(shell dpkg-parsechangelog | egrep '^Version:' |
cut -f 2 -d ' ' | cut -f 1 -d '-')
DEB_DH_MAKESHLIBS_ARGS_libpanel-applet2-0 := -V"libpanel-applet2-0 (>=
$(DEB_NOREVISION_VERSION))"
DEB_SHLIBDEPS_LIBRARY_gnome-panel:= libpanel-applet2-0
DEB_SHLIBDEPS_INCLUDE_gnome-panel := debian/libpanel-applet2-0/usr/lib/



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

binary-install/gnome-panel-data::
    chmod a+x debian/gnome-panel-data/etc/menu-methods/gnome-panel-data
    find debian/gnome-panel-data/usr/share -type f -exec chmod -R a-x {} \;

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

clean::
    # GNOME Team 'uploaders.mk' should not override this behavior, here is a
workarround :
    sed -i "s/@cdbs@/$(CDBS_BUILD_DEPENDS)/g" debian/control
```

Python Example

This example is from the package pmock. It builds Python modules for version 2.3 and 2.4 as well as a metapackage without writing any custom rules.

debian/control:

```
Source: pmock
Section: python
Priority: optional
Maintainer: Jan Alonzo <jmalonzo@unpluggable.com>
Build-Depends: debhelper (>= 4.1.67), cdbs, python2.3-dev, python2.4-dev,
python-dev (>= 2.3)
Standards-Version: 3.6.1.1

Package: python-pmock
Architecture: all
Depends: ${python:Depends}, python (>= 2.3), python (<< 2.5)
Description: Python module for unit testing using mock objects
 Mock Objects is a test-first development process for building object-
oriented
 software and a generic unit testing framework that supports that process.
 .
 This package allows you to use Mock Objects for unit testing Python
programs.
 .
 This is a dependency package which selects Debian's default Python version.
 .
  Homepage: http://pmock.sourceforge.net

Package: python2.3-pmock
Architecture: all
Depends: ${python:Depends}, python2.3
Description: Python module for unit testing using mock objects
 Mock Objects is a test-first development process for building object-
oriented
 software and a generic unit testing framework that supports that process.
 .
 This package allows you to use Mock Objects for unit testing Python
programs.
 .
  Homepage: http://pmock.sourceforge.net

Package: python2.4-pmock
Architecture: all
Depends: ${python:Depends}, python2.4
Description: Python module for unit testing using mock objects
 Mock Objects is a test-first development process for building object-
oriented
```

```
 software and a generic unit testing framework that supports that process.
 .
 This package allows you to use Mock Objects for unit testing Python
programs.
 .
  Homepage: http://pmock.sourceforge.net
```

debian/rules:

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

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

Makefile + Dpatch example

This example is from the package apg.

debian/control.in:

```
Source: apg
Section: admin
Priority: optional
Maintainer: Marc Haber <mh+debian-packages@zugschlus.de>
Build-Depends: @cdbs@
Standards-Version: 3.6.1

Package: apg
Architecture: any
Depends: ${shlibs:Depends}
Description: Automated Password Generator - Standalone version
 APG (Automated Password Generator) is the tool set for random
 password generation. It generates some random words of required type
 and prints them to standard output. This binary package contains only
 the standalone version of apg.
 Advantages:
  * Built-in ANSI X9.17 RNG (Random Number Generator)(CAST/SHA1)
  * Built-in password quality checking system (now it has support for Bloom
    filter for faster access)
  * Two Password Generation Algorithms:
     1. Pronounceable Password Generation Algorithm (according to NIST
        FIPS 181)
     2. Random Character Password Generation Algorithm with 35
        configurable modes of operation
  * Configurable password length parameters
  * Configurable amount of generated passwords
```

```
  * Ability to initialize RNG with user string
  * Support for /dev/random
  * Ability to crypt() generated passwords and print them as additional
output.
  * Special parameters to use APG in script
  * Ability to log password generation requests for network version
  * Ability to control APG service access using tcpd
  * Ability to use password generation service from any type of box (Mac,
    WinXX, etc.) that connected to network
  * Ability to enforce remote users to use only allowed type of password
    generation
 The client/server version of apg has been deliberately omitted.
 .
 Upstream URL: http://www.adel.nursat.kz/apg/download.shtml
```

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

# bug #284231
unpatch: deapply-dpatches
```

(Be advised that bug #284231 has been fixed in the meantime.)

Perl Examples

This example is from the package `libmidi-perl`. It builds a Perl module.

`debian/control`:

```
Source: libmidi-perl
Section: interpreters
Priority: optional
Build-Depends: cdbs (>= 0.4.4), debhelper (>= 4.1.0), perl (>= 5.8.0-7)
Maintainer: Mario Lang <mlang@debian.org>
Standards-Version: 3.5.10

Package: libmidi-perl
Architecture: all
Depends: ${perl:Depends}
Description: read, compose, modify, and write MIDI files in Perl
 This suite of Perl modules provides routines for reading, composing,
 modifying, and writing MIDI files.
```

`debian/rules`:

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

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

This example is from the package `libgd-graph-perl`. It illustrates the occasional need to set variables and add customized rules.

`debian/control`:

```
Source: libgd-graph-perl
Section: perl
Priority: extra
Maintainer: Jonas Smedegaard <dr@jones.dk>
Standards-Version: 3.6.1
Build-Depends-indep: cdbs, debhelper (>= 4.1), perl (>= 5.6.0-16), libgd-
text-perl (>= 0.80), imagemagick, dh-buildinfo

Package: libgd-graph-perl
Architecture: all
Depends: libgd-text-perl (>= 0.80)
Description:  Graph Plotting Module for Perl 5
 GD::Graph is a perl5 module to create charts using the GD module.
```

```
 The following classes for graphs with axes are defined:
 .
 GD::Graph::lines - Create a line chart.
 GD::Graph::bars - Create a bar chart.
 GD::Graph::points - Create an chart, displaying the data as points.
 GD::Graph::linespoints - Combination of lines and points.
 GD::Graph::area - Create a graph, representing the data as areas under a
     line.
 GD::Graph::mixed - Create a mixed type graph, any combination of the
     above. At the moment this is fairly limited. Some of
     the options that can be used with some of the
     individual graph types won't work very well. Multiple
     bar graphs in a mixed graph won't display very nicely.
 GD::Graph::pie - Create a pie chart.
```

debian/rules:

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

# Put perlmodule.mk last to dh_clean temporary files not in MANIFEST
include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/class/perlmodule.mk

DEB_INSTALL_EXAMPLES_libgd-graph-perl := samples Dustismo_Sans.ttf

# Upstream says creating samples is a better test so do both
DEB_MAKE_CHECK_TARGET := test samples

# Clean explicitly, as upstream make target "clean" in samples is broken
clean::
    rm -f $(CURDIR)/samples/sample*.png $(CURDIR)/samples/sample*.gif
$(CURDIR)/samples/logo.gif

# Add build info
common-binary-post-install-indep::
    dh_buildinfo
```

Ant Example

This example is from the package `jline`. Here you can see how to use the Ant class and set up rules that install the package in a policy-conforming way.

debian/control:

```
Source: jline
```

```
Section: libs
Priority: optional
Maintainer: Debian Java Maintainers <pkg-java-
maintainers@lists.alioth.debian.org>
Uploaders: Varun Hiremath <varun@debian.org>, Kumar Appaiah
<akumar@ee.iitm.ac.in>,
 Torsten Werner <twerner@debian.org>
Build-Depends: cdbs, debhelper (>= 5), default-jdk, ant
Build-Depends-Indep: maven-repo-helper, junit, openjdk-6-doc
Standards-Version: 3.8.2
Homepage: http://jline.sourceforge.net/
Vcs-Svn: svn://svn.debian.org/svn/pkg-java/trunk/jline
Vcs-Browser: http://svn.debian.org/wsvn/pkg-java/trunk/jline

Package: libjline-java
Section: libs
Architecture: all
Depends: ${misc:Depends}, default-jre-headless | java2-runtime-headless |
java1-runtime-headless
Suggests: libjline-java-doc
Description: Java library for handling console input
 JLine is a 100% pure Java library for reading and editing console input.
 It is similar in functionality to BSD editline and GNU readline.  People
 familiar with the readline/editline capabilities for modern shells will
 find most of the command editing features of JLine to be familiar.

Package: libjline-java-doc
Section: doc
Architecture: all
Depends: ${misc:Depends}, openjdk-6-doc | classpath-doc
Suggests: libjline-java
Description: documentation for JLine
 JLine is a 100% pure Java library for reading and editing console input.
 It is similar in functionality to BSD editline and GNU readline.  People
 familiar with the readline/editline capabilities for modern shells will
 find most of the command editing features of JLine to be familiar.
 .
 This package contains the documentation for JLine.
```

debian/rules

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

include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/class/ant.mk

PACKAGE              := $(DEB_SOURCE_PACKAGE)
VERSION              := $(DEB_UPSTREAM_VERSION)
JAVA_HOME                    := /usr/lib/jvm/default-java
DEB_JARS                     := junit
```

```
DEB_ANT_BUILDFILE        := debian/build.xml
DEB_ANT_BUILD_TARGET := jar javadoc

DEB_INSTALL_EXAMPLES_libjline-java-doc = jline-demo.jar

binary-post-install/lib$(PACKAGE)-java::
        mh_installpoms -plib$(PACKAGE)-java
        mh_installjar -plib$(PACKAGE)-java -l debian/pom.xml jline.jar

clean::
        -rm -rf debian/tmp

get-orig-source:
        -uscan --upstream-version 0

get-orig-pom:
        wget -O debian/pom.xml
http://repository.sonatype.org/service/local/repositories/central/content/jl
ine/jline/$(VERSION)/jline-$(VERSION).pom
```

Chapter 6. Conclusion

CDBS solves most common problems in building Debian packages, and it is very pleasant to use. More and more Debian packagers are using it, not because they are obliged to, but because they tasted and found it could improve their packages and avoid losing time on constantly reinventing silly and complicated rules.

CDBS is not perfect. It is not yet capable of handling very complicated situations, like packages where multiple C or C++ builds with different options and/or patches are required, but this only affects a very small number of packages. These limitations are planned to be solved in CDBS2, which is work in progress.

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

This document was originally written by Marc Dequènes and Arnaud Patard with the following revision history:

Revision History	
Revision 0.1.0	2005-04-03
First Public Release (for CDBS V0.4.27-3)	
Revision 0.1.1	2005-06-07

Updated for CDBS V0.4.30 (perl class build dependency management, cdbs-edit-patch script)
Revision 0.1.2 2005-07-05
Added DEB_CONFIGURE_SCRIPT_ENV usage warning, fixed typo.

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

Special thanks to GuiHome for helping by reviewing 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).