`dpkg` programmers' manual

Ian Jackson `<ijackson@gnu.ai.mit.edu>`

version 0.2.0.0 (dpkg 1.3.7), 23 August 1996

0.1. Abstract

This manual describes the technical aspects of creating Debian binary and source packages. It also documents the interface between `dselect` and its access method scripts. It does not deal with the Debian Project policy requirements, and it assumes familiarity with `dpkg`'s functions from the system administrator's perspective.

0.2. Contents

0.3. Copyright Notice

1. Introduction and scope of this manual

`dpkg` is a suite of programs for creating binary package files and installing and removing them on Unix systems.[1]

The binary packages are designed for the management of installed executable programs (usually compiled binaries) and their associated data, though source code examples and documentation are provided as part of some packages.

This manual describes the technical aspects of creating Debian binary packages (`.deb` files). It documents the behaviour of the package management programs `dpkg`, `dselect` et al. and and the way they interact with packages.

It also documents the interaction between `dselect`'s core and the access method scripts it uses to actually install the selected packages, and describes how to create a new access method.

This manual does not go into detail about the options and usage of the package building and installation tools. It should therefore be read in conjuction with those programs' manpages.

The utility programs which are provided with `dpkg` for managing various system configuration and similar issues, such as `update-rc.d` and `install-info`, are not described in detail here - please see their manpages.

It does *not* describe the policy requirements imposed on Debian packages, such as the permissions on files and directories, documentation requirements, upload procedure, and so on. You should see the Debian packaging policy manual for these details. (Many of them will probably turn out to be helpful even if you don't plan to upload your package and make it available as part of the distribution.)

It is assumed that the reader is reasonably familiar with the `dpkg` System Administrators' manual. Unfortunately this manual does not yet exist.

The Debian version of the FSF's GNU hello program is provided as an example for people wishing to create Debian packages.

Note that this document is still a draft!

[1] `dpkg` is targetted primarily at Debian GNU/Linux, but may work on or be ported to other systems.

2. Binary packages

The binary package has two main sections. The first part consists of various control information files and scripts used by `dpkg` when installing and removing. See 'Package control information files', page 5.

The second part is an archive (currently a `tar` archive) containing files and directories to be installed.

In the future binary packages may also contain other components, such as checksums and digital signatures.

2.1. Creating package files - `dpkg-deb`

All manipulation of binary package files is done by `dpkg-deb`; it's the only program that has knowledge of the format. (`dpkg-deb` may be invoked by calling `dpkg`, as `dpkg` will spot that the options requested are appropriate to `dpkg-deb` and invoke that instead with the same arguments.)

In order to create a binary package you must make a directory tree which contains all the files and directories you want to have in the filesystem data part of the package. In Debian-format source packages this directory is usually `debian/tmp`, relative to the top of the package's source tree.

They should have the locations (relative to the root of the directory tree you're constructing) ownerships and permissions which you want them to have on the system when they are installed.

With current versions of `dpkg` the uid/username and gid/groupname mappings for the users and groups being used should be the same on the system where the package is built and the one where it is installed.

You need to add one special directory to the root of the miniature filesystem tree you're creating: `DEBIAN`. It should contain the control information files, notably the binary package control file (see 'The main control information file: `control`', page 6).

The `DEBIAN` directory will not appear in the filesystem archive of the package, and so won't be installed by `dpkg` when the package is installed.

When you've prepared the package, you should invoke:

    dpkg --build *directory*

This will build the package in *directory*`.deb`. (`dpkg` knows that `--build` is a `dpkg-deb` option, so it invokes `dpkg-deb` with the same arguments to build the package.)

See the manpage `dpkg-deb(8)` for details of how to examine the contents of this newly-created file. You may find the output of following commands enlightening:

    dpkg-deb --info *filename*.deb
    dpkg-deb --contents *filename*.deb

2.2. Package control information files

The control information portion of a binary package is a collection of files with names known to `dpkg`. It will treat the contents of these files specially - some of them contain information used by `dpkg` when installing or removing the package; others are scripts which the package maintainer wants `dpkg` to run.

It is possible to put other files in the package control area, but this is not generally a good idea (though they will largely be ignored).

Here is a brief list of the control info files supported by `dpkg` and a summary of what they're used for.

`control`
: This is the key description file used by `dpkg`. It specifies the package's name and version, gives its description for the user, states its relationships with other packages, and so forth. See 'The main control information file: `control`', page 6.

  It is usually generated automatically from information in the source package by the `dpkg-gencontrol` program, and with assistance from `dpkg-shlibdeps`. See 'Tools for processing source packages', page 8.

`postinst`, `preinst`, `postrm`, `prerm`
: These are exectuable files (usually scripts) which `dpkg` runs during installation, upgrade and removal of packages. They allow the package to deal with matters which are particular to that package or require more complicated processing than that provided by `dpkg`. Details of when and how they are called are in 'Package maintainer scripts and installation procedure', page 28.

  It is very important to make these scripts itempotent.[1] This is so that if an error occurs, the user interrupts `dpkg` or some other unforeseen circumstance happens you don't leave the user with a badly-broken package.

  The maintainer scripts are guaranteed to run with a controlling terminal and can interact with the user. If they need to prompt for passwords, do full-screen interaction or something similar you should do these things to and from `/dev/tty`, since `dpkg` will at some point redirect scripts' standard input and output so that it can log the installation process. Likewise, because these scripts may be executed with standard output redirected into a pipe for logging purposes, Perl scripts should set unbuffered output by setting `$|=1` so that the output is printed immediately rather than being buffered.

  Each script should return a zero exit status for success, or a nonzero one for failure.

`conffiles`
: This file contains a list of configuration files which are to be handled automatically by `dpkg` (see 'Configuration file handling', page 42). Note that not necessarily every configuration file should be listed here.

`shlibs`
: This file contains a list of the shared libraries supplied by the package, with dependency

[1] That means that if it runs successfully or fails and then you call it again it doesn't bomb out, but just ensures that everything is the way it ought to be.

details for each. This is used by `dpkg-shlibdeps` when it determines what dependencies are required in a package control file.

Each line is of the form:
library–name version–or–soname dependencies ...

library-name is the name of the shared library, for example `libc5`.

version-or-soname is the soname of the library - ie, the thing that must exactly match for the library to be recognised by `ld.so`. Usually this is major version number of the library.

dependencies has the same syntax as a dependency field in a binary package control file. It should give details of which package(s) are required to satisfy a binary built against the version of the library contained in the package. See ‘Syntax of relationship fields’, page 36.

For example, if the package `foo` contains `libfoo.so.1.2.3`, where the soname of the library is `libfoo.so.1`, and the first version of the package which contained a minor number of at least `2.3` was *1.2.3-1*, then the package’s *shlibs* could say:

```
libfoo 1        foo (>= 1.2.3-1)
```

The version-specific dependency is to avoid warnings from `ld.so` about using older shared libraries with newer binaries.

2.3. The main control information file: `control`

The most important control information file used by `dpkg` when it installs a package is `control`. It contains all the package’s ‘vital statistics’.

The binary package control files of packages built from Debian sources are made by a special tool, `dpkg-gencontrol`, which reads `debian/control` and `debian/changelog` to find the information it needs. See ‘Source packages’, page 8 for more details.

The fields in binary package control files are:
- `Package` (mandatory)
- `Version` (mandatory)
- `Architecture` (mandatory)[1]
- `Depends`, `Provides` et al.
- `Essential`
- `Maintainer`
- `Section`, `Priority`
- `Source`
- `Description`
- `Installed-Size`

A description of the syntax of control files and the purpose of these fields is available in ‘Control files and their fields’, page 19.

[1]This field should appear in all packages, though `dpkg` doesn’t require it yet so that old packages can still be installed.

3. Source packages

The Debian binary packages in the distribution are generated from Debian sources, which are in a special format to assist the easy and automatic building of binaries.

3.1. Tools for processing source packages

Various tools are provided for manipulating source packages; they pack and unpack sources and help build of binary packages and help manage the distribution of new versions.

They are introduced and typical uses described here; see `dpkg-source(1)` for full documentation about their arguments and operation.

For examples of how to construct a Debian source package, and how to use those utilities that are used by Debian source packages, please see the `hello` example package.

3.1.1. `dpkg-source` - packs and unpacks Debian source packages

This program is frequently used by hand, and is also called from package-independent automated building scripts such as `dpkg-buildpackage`.

To unpack a package it is typically invoked with
    `dpkg-source -x` *.../path/to/filename*`.dsc`
with the *filename*`.tar.gz` and *filename*`.diff.gz` (if applicable) in the same directory. It unpacks into *package*-*version*, and if applicable *package*-*version*`.orig`, in the current directory.

To create a packed source archive it is typically invoked:
    `dpkg-source -b` *package*-*version*
This will create the `.dsc`, `.tar.gz` and `.diff.gz` (if appropriate) in the current directory. `dpkg-source` does not clean the source tree first - this must be done separately if it is required.

See also ‘Source packages as archives’, page 16.

3.1.2. `dpkg-buildpackage` - overall package-building control script

`dpkg-buildpackage` is a script which invokes `dpkg-source`, the `debian/rules` targets `clean`, `build` and `binary`, `dpkg-genchanges` and `pgp` to build a signed source and binary package upload.

It is usually invoked by hand from the top level of the built or unbuilt source directory. It may be invoked with no arguments; useful arguments include:

`-uc`, `-us`
    Do not PGP-sign the `.changes` file or the source package `.dsc` file, respectively.

`-p`*pgp-command*
    Invoke *pgp-command* instead of finding `pgp` on the `PATH`. *pgp-command* must behave just like `pgp`.

`-r`*root-command*
    When root privilege is required, invoke the command *root-command*. *root-command* should invoke its first argument as a command, from the `PATH` if necessary, and pass its second

and subsequent arguments to the command it calls. If no *root-command* is supplied then *dpkg-buildpackage* will take no special action to gain root privilege, so that for most packages it will have to be invoked as root to start with.

-b, **-B**
Two types of binary-only build and upload - see **dpkg-source(1)**.

3.1.3. `dpkg-gencontrol` - generates binary package control files

This program is usually called from **debian/rules** (see 'The Debianised source tree', page 11) in the top level of the source tree.

This is usually done just before the files and directories in the temporary directory tree where the package is being built have their permissions and ownerships set and the package is constructed using **dpkg-deb**[1].

dpkg-gencontrol must be called after all the files which are to go into the package have been placed in the temporary build directory, so that its calculation of the installed size of a package is correct.

It is also necessary for **dpkg-gencontrol** to be run after **dpkg-shlibdeps** so that the variable substitutions created by **dpkg-shlibdeps** in **debian/substvars** are available.

For a package which generates only one binary package, and which builds it in **debian/tmp** relative to the top of the source package, it is usually sufficient to call:

```
dpkg-gencontrol
```

Sources which build several binaries will typically need something like:

dpkg-gencontrol -Pdebian/tmp-*pkg* **-p***package*

The **-P** tells **dpkg-gencontrol** that the package is being built in a non-default directory, and the **-p** tells it which package's control file should be generated.

dpkg-gencontrol also adds information to the list of files in **debian/files**, for the benefit of (for example) a future invocation of **dpkg-genchanges**.

3.1.4. `dpkg-shlibdeps` - calculates shared library dependencies

This program is usually called from **debian/rules** just before **dpkg-gencontrol** (see 'The Debianised source tree', page 11), in the top level of the source tree.

Its arguments are executables[2] for which shared library dependencies should be included in the binary package's control file.

If some of the executable(s) shared libraries should only warrant a **Recommends** or **Suggests**, or if some warrant a **Pre-Depends**, this can be achieved by using the **-d***dependency-field* option before those executable(s). (Each **-d** option takes effect until the next **-d**.)

dpkg-shlibdeps does not directly cause the output control file to be modified. Instead by default it adds to the **debian/substvars** file variable settings like **shlibs:Depends**. These variable settings

[1]This is so that the control file which is produced has the right permissions

[2]They may be specified either in the locations in the source tree where they are created or in the locations in the temporary build tree where they are installed prior to binary package creation.

must be referenced in dependency fields in the appropriate per-binary-package sections of the source control file.

For example, the `procps` package generates two kinds of binaries, simple C binaries like `ps` which require a predependency and full-screen ncurses binaries like `top` which require only a recommendation. It can say in its `debian/rules`:

```
dpkg-shlibdeps -dPre-Depends ps -dRecommends top
```

and then in its main control file `debian/control`:

```
...
Package: procps
Pre-Depends: ${shlibs:Pre-Depends}
Recommends: ${shlibs:Recommends}
...
```

Sources which produce several binary packages with different shared library dependency requirements can use the `-p`*varnameprefix* option to override the default `shlib:` prefix (one invocation of `dpkg-shlibdeps` per setting of this option). They can thus produce several sets of dependency variables, each of the form *varnameprefix*`:`*dependencyfield*, which can be referred to in the appropriate parts of the binary package control files.

3.1.5. `dpkg-distaddfile` - adds a file to `debian/files`

Some packages' uploads need to include files other than the source and binary package files.

`dpkg-distaddfile` adds a file to the `debian/files` file so that it will be included in the `.changes` file when `dpkg-genchanges` is run.

It is usually invoked from the `binary` target of `debian/rules`:

`dpkg-distaddfile` *filename section priority*

The *filename* is relative to the directory where `dpkg-genchanges` will expect to find it - this is usually the directory above the top level of the source tree. The `debian/rules` target should put the file there just before or just after calling `dpkg-distaddfile`.

The *section* and *priority* are passed unchanged into the resulting `.changes` file. See '`Section` and `Priority`', page 21.

3.1.6. `dpkg-genchanges` - generates a `.changes` upload control file

This program is usually called by package-independent automatic building scripts such as `dpkg-buildpackage`, but it may also be called by hand.

It is usually called in the top level of a built source tree, and when invoked with no arguments will print out a straightforward `.changes` file based on the information in the source package's changelog and control file and the binary and source packages which should have been built.

3.1.7. `dpkg-parsechangelog` - produces parsed representation of a changelog

This program is used internally by `dpkg-source` et al. It may also occasionally be useful in `debian/rules` and elsewhere. It parses a changelog, `debian/changelog` by default, and prints a control-file format representation of the information in it to standard output.

3.2. The Debianised source tree

The source archive scheme described later is intended to allow a Debianised source tree with some associated control information to be reproduced and transported easily. The Debianised source tree is a version of the original program with certain files added for the benefit of the Debianisation process, and with any other changes required made to the rest of the source code and installation scripts.

The extra files created for Debian are in the subdirectory `debian` of the top level of the Debianised source tree. They are described below.

3.2.1. `debian/rules` - the main building script

This file is an executable makefile, and contains the package-specific recipies for compiling the package and building binary package(s) out of the source.

It must start with the line `#!/usr/bin/make -f`, so that it can be invoked by saying its name rather than invoking `make` explicitly.

The targets which are required to be present are:

`build`
: This should perform all non-interactive configuration and compilation of the package. If a package has an interactive pre-build configuration routine, the Debianised source package should be built after this has taken place, so that it can be built without rerunning the configuration.

  For some packages, notably ones where the same source tree is compiled in different ways to produce two binary packages, the `build` target does not make much sense. For these packages it is good enough to provide two (or more) targets (`build-a` and `build-b` or whatever) for each of the ways of building the package, and a `build` target that does nothing. The `binary` target will have to build the package in each of the possible ways and make the binary package out of each.

  The `build` target must not do anything that might require root privilege.

  The `build` target may need to run `clean` first - see below.

  When a package has a configuration routine that takes a long time, or when the makefiles are poorly designed, or when `build` needs to run `clean` first, it is a good idea to `touch build` when the build process is complete. This will ensure that if `debian/rules build` is run again it will not rebuild the whole program.

`binary`, `binary-arch`, `binary-indep`
: The `binary` target should be all that is necessary for the user to build the binary package. It is split into two parts: `binary-arch` builds the packages' output files which are specific to a particular architecture, and `binary-indep` builds those which are not.

  `binary` should usually be a target with no commands which simply depends on `binary-arch` and `binary-indep`.

  Both `binary-*` targets should depend on the `build` target, above, so that the package is built if it has not been already. It should then create the relevant binary package(s), using `dpkg-gencontrol` to make their control files and `dpkg-deb` to build them and place them in the parent of the top level directory.

If one of the `binary-*` targets has nothing to do (this will be always be the case if the source generates only a single binary package, whether architecture-dependent or not) it *must* still exist, but should always succeed.

'Binary packages', page 4 describes how to construct binary packages.

The `binary` targets must be invoked as root.

`clean`
This should undo any effects that the `build` and `binary` targets may have had, except that it should leave alone any output files created in the parent directory by a run of `binary`.

If a `build` file is touched at the end of the `build` target, as suggested above, it must be removed as the first thing that `clean` does, so that running `build` again after an interrupted `clean` doesn't think that everything is already done.

The `clean` target must be invoked as root if `binary` has been invoked since the last `clean`, or if `build` has been invoked as root (since `build` may create directories, for example).

`get-orig-source`
This target fetches the most recent version of the original source package from a canonical archive site (via FTP or WWW, for example), does any necessary rearrangement to turn it into the original source tarfile format described below, and leaves it in the current directory.

This target may be invoked in any directory, and should take care to clean up any temporary files it may have left.

This target is optional, but providing it if possible is a good idea.

The `build`, `binary` and `clean` targets must be invoked with a current directory of the package's top-level directory.

Additional targets may exist in `debian/rules`, either as published or undocumented interfaces or for the package's internal use.

3.2.2. `debian/control`

This file contains version-independent details about the source package and about the binary packages it creates.

It is a series of sets of control fields, each syntactically similar to a binary package control file. The sets are separated by one or more blank lines. The first set is information about the source package in general; each subsequent set describes one binary package that the source tree builds.

The syntax and semantics of the fields are described below in 'Control files and their fields', page 19.

The general (binary-package-independent) fields are:
- `Source` (mandatory)
- `Maintainer`
- `Section` and `Priority` (classification, mandatory)
- `Standards-Version`

The per-binary-package fields are:
- `Package` (mandatory)
- `Architecture` (mandatory)
- `Description`
- `Section` and `Priority` (classification)
- `Essential`
- `Depends` et al. (package interrelationships)

These fields are used by `dpkg-gencontrol` to generate control files for binary packages (see below), by `dpkg-genchanges` to generate the `.changes` file to accompany the upload, and by `dpkg-source` when it creates the `.dsc` source control file as part of a source archive.

The fields here may contain variable references - their values will be substituted by `dpkg-gencontrol`, `dpkg-genchanges` or `dpkg-source` when they generate output control files. See '`debian/substvars` and variable substitutions', page 15 for details.

3.2.2.1. User-defined fields

Additional user-defined fields may be added to the source package control file. Such fields will be ignored, and not copied to (for example) binary or source package control files or upload control files.

If you wish to add additional unsupported fields to these output files you should use the mechanism described here.

Fields in the main source control information file with names starting `X`, followed by one or more of the letters `BCS` and a hyphen `-`, will be copied to the output files. Only the part of the field name after the hyphen will be used in the output file. Where the letter `B` is used the field will appear in binary package control files, where the letter `S` is used in source package control files and where `C` is used in upload control (`.changes`) files.

For example, if the main source information control file contains the field

```
XBS-Comment: I stand between the candle and the star.
```

then the binary and source package control files will contain the field

```
Comment: I stand between the candle and the star.
```

3.2.3. `debian/changelog`

This file records the changes to the Debian-specific parts of the package[1].

It has a special format which allows the package building tools to discover which version of the package is being built and find out other release-specific information.

That format is a series of entries like this:

[1]Though there is nothing stopping an author who is also the Debian maintainer from using it for all their changes, it will have to be renamed if the Debian and upstream maintainers become different people.

```
package (version) distribution(s); urgency=urgency

  * change details
    more change details
  * even more change details

 -- maintainer name and email address  date
```

package and *version* are the source package name and version number. *distribution(s)* lists the distributions where this version should be installed when it is uploaded - it is copied to the `Distribution` field in the `.changes` file. See ‘`Distribution`’, page 23.

urgency is the value for the `Urgency` field in the `.changes` file for the upload. See ‘`Urgency`’, page 23. It is not possible to specify an urgency containing commas; commas are used to separate *keyword*`=`*value* settings in the `dpkg` changelog format (though there is currently only one useful *keyword*, `urgency`).

The change details may in fact be any series of lines starting with at least two spaces, but conventionally each change starts with an asterisk and a separating space and continuation lines are indented so as to bring them in line with the start of the text above. Blank lines may be used here to separate groups of changes, if desired.

The maintainer name and email address should *not* necessarily be those of the usual package maintainer. They should be the details of the person doing *this* version. The information here will be copied to the `.changes` file, and then later used to send an acknowledgement when the upload has been installed.

The *date* should be in RFC822 format; it should include the timezone specified numerically, with the timezone name or abbreviation optionally present as a comment.

The first ‘title’ line with the package name should start at the left hand margin; the ‘trailer’ line with the maintainer and date details should be preceded by exactly one space. The maintainer details and the date must be separated by exactly two spaces.

An Emacs mode for editing this format is available: it is called `debian-changelog-mode`. You can have this mode selected automatically when you edit a Debian changelog by adding a local variables clause to the end of the changelog.

3.2.3.1. Defining alternative changelog formats

It is possible to use a different format to the standard one, by providing a parser for the format you wish to use.

In order to have `dpkg-parsechangelog` run your parser, you must include a line within the last 40 lines of your file matching the Perl regular expression:

```
\schangelog-format:\s+([0-9a-z]+)\W
```

The part in parentheses should be the name of the format.

If such a line exists then `dpkg-parsechangelog` will look for the parser as `/usr/lib/dpkg/parsechangelog/`*format-name* or `/usr/local/lib/dpkg/parsechangelog/`*format-name*; it is an error for it not to find it, or for it not to be an executable program. The default changelog format is `dpkg`, and a parser for it is provided with the `dpkg` package.

The parser will be invoked with the changelog open on standard input at the start of the file. It should read the file (it may seek if it wishes) to determine the information required and return the parsed information to standard output in the form of a series of control fields in the standard format. By default it should return information about only the most recent version in the changelog; it should accept a `-v`*version* option to return changes information from all versions present *strictly after version*, and it should then be an error for *version* not to be present in the changelog.

The fields are:
- `Source`
- `Version` (mandatory)
- `Distribution` (mandatory)
- `Urgency` (mandatory)
- `Maintainer` (mandatory)
- `Date`
- `Changes` (mandatory)

If several versions are being returned (due to the use of `-v`), the urgency value should be of the highest urgency code listed at the start of any of the versions requested followed by the concatenated (space-separated) comments from all the versions requested; the maintainer, version, distribution and date should always be from the most recent version.

For the format of the `Changes` field see '`Changes`', page 24.

If the changelog format which is being parsed always or almost always leaves a blank line between individual change notes these blank lines should be stripped out, so as to make the resulting output compact.

If the changelog format does not contain date or package name information this information should be omitted from the output. The parser should not attempt to synthesise it or find it from other sources.

If the changelog does not have the expected format the parser should exit with a nonzero exit status, rather than trying to muddle through and possibly generating incorrect output.

A changelog parser may not interact with the user at all.

3.2.4. `debian/substvars` and variable substitutions

When `dpkg-gencontrol`, `dpkg-genchanges` and `dpkg-source` generate control files they do variable substitutions on their output just before writing it. Variable substitutions have the form `${`*variable-name*`}`. The optional file `debian/substvars` contains variable substitutions to be used; variables can also be set directly from `debian/rules` using the `-V` option to the source packaging commands, and certain predefined variables are available.

The file may be a static part of the source archive, or generated and modified dynamically by `debian/rules` targets. In the latter case it must be removed by the `clean` target.

See `dpkg-source(1)` for full details about source variable substitutions, including the format of `debian/substvars`.

3.2.5. `debian/files`

This file is not a permanent part of the source tree; it is used while building packages to record which files are being generated. `dpkg-genchanges` uses it when it generates a `.changes` file.

It should not exist in a shipped source package, and so it (and any backup files or temporary files such as `files.new`[1]) should be removed by the `clean` target. It may also be wise to ensure a fresh start by emptying or removing it at the start of the `binary` target.

`dpkg-gencontrol` adds an entry to this file for the `.deb` file that will be created by `dpkg-deb` from the control file that it generates, so for most packages all that needs to be done with this file is to delete it in `clean`.

If a package upload includes files besides the source package and any binary packages whose control files were made with `dpkg-gencontrol` then they should be placed in the parent of the package's top-level directory and `dpkg-distaddfile` should be called to add the file to the list in `debian/files`.

3.2.6. `debian/tmp`

This is the canonical temporary location for the construction of binary packages by the `binary` target. The directory `tmp` serves as the root of the filesystem tree as it is being constructed (for example, by using the package's upstream makefiles install targets and redirecting the output there), and it also contains the `DEBIAN` subdirectory. See 'Creating package files - `dpkg-deb`', page 4.

If several binary packages are generated from the same source tree it is usual to use several `debian/tmp`*something* directories, for example `tmp-a` or `tmp-doc`.

Whatever `tmp` directories are created and used by `binary` must of course be removed by the `clean` target.

3.3. Source packages as archives

As it exists on the FTP site, a Debian source package consists of three related files. You must have the right versions of all three to be able to use them.

Debian source control file - `.dsc`

This file contains a series of fields, identified and separated just like the fields in the control file of a binary package. The fields are listed below; their syntax is described above, in 'Control files and their fields', page 19.

- `Source`
- `Version`
- `Maintainer`
- `Binary`
- `Architecture`
- `Standards-Version`
- `Files`

The source package control file is generated by `dpkg-source` when it builds the source archive, from other files in the source package, described above. When unpacking it is checked against the files and directories in the other parts of the source package, as described below.

[1] `files.new` is used as a temporary file by `dpkg-gencontrol` and `dpkg-distaddfile` - they write a new version of `files` here before renaming it, to avoid leaving a corrupted copy if an error occurs

Original source archive - *package_upstream-version*`.orig.tar.gz`
: This is a compressed (with `gzip -9`) `tar` file containing the source code from the upstream authors of the program. The tarfile unpacks into a directory *package-upstream-version*`.orig`, and does not contain files anywhere other than in there or in its subdirectories.

Debianisation diff - *package_version-revision*`.diff.gz`
: This is a unified context diff (`diff -u`) giving the changes which are required to turn the original source into the Debian source. These changes may only include editing and creating plain files. The permissions of files, the targets of symbolic links and the characteristics of special files or pipes may not be changed and no files may be removed or renamed.

  All the directories in the diff must exist, except the `debian` subdirectory of the top of the source tree, which will be created by `dpkg-source` if necessary when unpacking.

  The `dpkg-source` program will automatically make the `debian/rules` file executable (see below).

If there is no original source code - for example, if the package is specially prepared for Debian or the Debian maintainer is the same as the upstream maintainer - the format is slightly different: then there is no diff, and the tarfile is named *package_version*`.tar.gz` and contains a directory *package-version*.

3.4. Unpacking a Debian source package without `dpkg-source`

`dpkg-source -x` is the recommended way to unpack a Debian source package. However, if it is not available it is possible to unpack a Debian source archive as follows:

1. Untar the tarfile, which will create a `.orig` directory.
2. Rename the `.orig` directory to *package-version*.
3. Create the subdirectory `debian` at the top of the source tree.
4. Apply the diff using `patch -p0`.
5. Untar the tarfile again if you want a copy of the original source code alongside the Debianised version.

It is not possible to generate a valid Debian source archive without using `dpkg-source`. In particular, attempting to use `diff` directly to generate the `.diff.gz` file will not work.

3.4.1. Restrictions on objects in source packages

The source package may not contain any device special files, sockets or setuid or setgid files.[1]

The source packaging tools manage the changes between the original and Debianised source using `diff` and `patch`. Turning the original source tree as included in the `.orig.tar.gz` into the debianised source must not involve any changes which cannot be handled by these tools. Problematic changes which cause `dpkg-source` to halt with an error when building the source package are:

- Adding or removing symbolic links, sockets or pipes.
- Changing the targets of symbolic links.
- Creating directories, other than `debian`.
- Changes to the contents of binary files.

---

[1] Setgid directories are allowed.

Changes which cause `dpkg-source` to print a warning but continue anyway are:

- Removing files, directories or symlinks.[1]

Changes which are not represented, but which are not detected by `dpkg-source`, are:

- Changing the permissions of files (other than `debian/rules`) and directories.

The `debian` directory and `debian/rules` are handled specially by `dpkg-source` - before applying the changes it will create the `debian` directory, and afterwards it will make `debian/rules` world-exectuable.

[1]Renaming a file is not treated specially - it is seen as the removal of the old file (which generates a warning, but is otherwise ignored), and the creation of the new one.

4. Control files and their fields

Many of the tools in the `dpkg` suite manipulate data in a common format, known as control files. Binary and source packages have control data as do the `.changes` files which control the installation of uploaded files, and `dpkg`'s internal databases are in a similar format.

4.1. Syntax of control files

A file consists of one or more paragraphs of fields. The paragraphs are separated by blank lines. Some control files only allow one paragraph; others allow several, in which case each paragraph often refers to a different package.

Each paragraph is a series of fields and values; each field consists of a name, followed by a colon and the value. It ends at the end of the line. Horizontal whitespace (spaces and tabs) may occur before or after the value and is ignored there; it is conventional to put a single space after the colon.

Some fields' values may span several lines; in this case each continuation line *must* start with a space or tab. Any trailing spaces or tabs at the end of individual lines of a field value are ignored.

Except where otherwise stated only a single line of data is allowed and whitespace is not significant in a field body. Whitespace may never appear inside names (of packages, architectures, files or anything else), version numbers or in between the characters of multi-character version relationships.

Field names are not case-sensitive, but it is usual to capitalise the fields using mixed case as shown below.

Blank lines, or lines consisting only of spaces and tabs, are not allowed within field values or between fields - that would mean a new paragraph.

It is important to note that there are several fields which are optional as far as `dpkg` and the related tools are concerned, but which must appear in every Debian package, or whose omission may cause problems. When writing the control files for Debian packages you *must* read the Debian policy manual in conjuction with the details below and the list of fields for the particular file.

4.2. List of fields

4.2.1. `Package`

The name of the binary package. Package names consist of the alphanumerics and `+ - .` (plus, minus and full stop).[1]

They must be at least two characters and must start with an alphanumeric. In current versions

[1]The characters `@ : = % _` (at, colon, equals, percent and underscore) used to be legal and are still accepted when found in a package file, but may not be used in new packages

of dpkg they are sort of case-sensitive[1]; use lowercase package names unless the package you're building (or referring to, in other fields) is already using uppercase.

4.2.2. `Version`

This lists the source or binary package's version number - see 'Version numbering', page 26.

4.2.3. `Architecture`

This is the architecture string; it is a single word for the CPU architecture.

`dpkg` will check the declared architecture of a binary package against its own compiled-in value before it installs it.

The special value `all` indicates that the package is architecture-independent.

In the main `debian/control` file in the source package, or in the source package control file `.dsc`, a list of architectures (separated by spaces) is also allowed, as is the special value `any`. A list indicates that the source will build an architecture-dependent package, and will only work correctly on the listed architectures. `any` indicates that though the source package isn't dependent on any particular architecture and should compile fine on any one, the binary package(s) produced are not architecture-independent but will instead be specific to whatever the current build architecture is.

In a `.changes` file the `Architecture` field lists the architecture(s) of the package(s) currently being uploaded. This will be a list; if the source for the package is being uploaded too the special entry `source` is also present.

The current build architecture can be determined using `dpkg --print-architecture`.[2] This value is automatically used by `dpkg-gencontrol` when building the control file for a binary package for which the source control information doesn't specify architecture `all`.

There is a separate option, `--print-installation-architecture`, for finding out what architecture `dpkg` is willing to install. This information is also in the output of `dpkg --version`.

4.2.4. `Maintainer`

The package maintainer's name and email address. The name should come first, then the email address inside angle brackets `<>` (in RFC822 format).

If the maintainer's name contains a full stop then the whole field will not work directly as an email address due to a misfeature in the syntax specified in RFC822; a program using this field as an address must check for this and correct the problem if necessary (for example by putting the name in round brackets and moving it to the end, and bringing the email address forward).

In a `.changes` file or parsed changelog data this contains the name and email address of the person responsible for the particular version in question - this may not be the package's usual maintainer.

---

[1]This is a bug.

[2]This actually invokes

```
gcc --print-libgcc-file-name
```

and parses and decomposes the output and looks the CPU type from the GCC configuration in a table in `dpkg`. This is so that it will work if you're cross-compiling.

This field is usually optional in as far as the `dpkg` are concerned, but its absence when building packages usually generates a warning.

4.2.5. `Source`

This field identifies the source package name.

In a main source control information or a `.changes` or `.dsc` file or parsed changelog data this may contain only the name of the source package.

In the control file of a binary package (or in a `Packages` file) it may be followed by a version number in parentheses.[1] This version number may be omitted (and is, by `dpkg-gencontrol`) if it has the same value as the `Version` field of the binary package in question. The field itself may be omitted from a binary package control file when the source package has the same name and version as the binary package.

4.2.6. Package interrelationship fields: `Depends`, `Pre-Depends`, `Recommends` `Suggests`, `Conflicts`, `Provides`, `Replaces`

These fields describe the package's relationships with other packages. Their syntax and semantics are described in 'Declaring relationships between packages', page 36.

4.2.7. `Description`

In a binary package `Packages` file or main source control file this field contains a description of the binary package, in a special format. See 'Descriptions of packages - the `Description` field', page 33 for details.

In a `.changes` file it contains a summary of the descriptions for the packages being uploaded. The part of the field before the first newline is empty; thereafter each line has the name of a binary package and the summary description line from that binary package. Each line is indented by one space.

4.2.8. `Essential`

This is a boolean field which may occur only in the control file of a binary package (or in the `Packages` file) or in a per-package fields paragraph of a main source control data file.

If set to `yes` then `dpkg` and `dselect` will refuse to remove the package (though it can be upgraded and/or replaced). The other possible value is `no`, which is the same as not having the field at all.

4.2.9. `Section` and `Priority`

These two fields classify the package. The `Priority` represents how important that it is that the user have it installed; the `Section` represents an application area into which the package has been classified.

When they appear in the `debian/control` file these fields give values for the section and priority subfields of the `Files` field of the `.changes` file, and give defaults for the section and priority of

[1]It is usual to leave a space after the package name if a version number is specified.

the binary packages.

The section and priority are represented, though not as separate fields, in the information for each file in the `Files` field of a `.changes` file. The section value in a `.changes` file is used to decide where to install a package in the FTP archive.

These fields are not used by by `dpkg` proper, but by `dselect` when it sorts packages and selects defaults. See the Debian policy manual for the priorities in use and the criteria for selecting the priority for a Debian package, and look at the Debian FTP archive for a list of currently in-use priorities.

These fields may appear in binary package control files, in which case they provide a default value in case the `Packages` files are missing the information. `dpkg` and `dselect` will only use the value from a `.deb` file if they have no other information; a value listed in a `Packages` file will always take precedence. By default `dpkg-genchanges` does not include the section and priority in the control file of a binary package - use the `-isp`, `-is` or `-ip` options to achieve this effect.

4.2.10. `Binary`

This field is a list of binary packages.

When it appears in the `.dsc` file it is the list of binary packages which a source package can produce. It does not necessarily produce all of these binary packages for every architecture. The source control file doesn't contain details of which architectures are appropriate for which of the binary packages.

When it appears in a `.changes` file it lists the names of the binary packages actually being uploaded.

The syntax is a list of binary packages separated by commas.[1] Currently the packages must be separated using only spaces in the `.changes` file.

4.2.11. `Installed-Size`

This field appears in the control files of binary packages, and in the `Packages` files. It gives the total amount of disk space required to install the named package.

The disk space is represented in kilobytes as a simple decimal number.

4.2.12. `Files`

This field contains a list of files with information about each one. The exact information and syntax varies with the context. In all cases the the part of the field contents on the same line as the field name is empty. The remainder of the field is one line per file, each line being indented by one space and containing a number of sub-fields separated by spaces.

In the `.dsc` (Debian source control) file each line contains the MD5 checksum, size and filename of the tarfile and (if applicable) diff file which make up the remainder of the source package.[2] The exact forms of the filenames are described in 'Source packages as archives', page 16.

[1] A space after each comma is conventional.

[2] That is, the parts which are not the `.dsc`.

In the `.changes` file this contains one line per file being uploaded. Each line contains the MD5 checksum, size, section and priority and the filename. The section and priority are the values of the corresponding fields in the main source control file - see '`Section` and `Priority`', page 21. If no section or priority is specified then `-` should be used, though section and priority values must be specified for new packages to be installed properly.

The special value `byhand` for the section in a `.changes` file indicates that the file in question is not an ordinary package file and must by installed by hand by the distribution maintainers. If the section is `byhand` the priority should be `-`.

If a new Debian revision of a package is being shipped and no new original source archive is being distributed the `.dsc` must still contain the `Files` field entry for the original source archive *package-upstream-version*`.orig.tar.gz`, but the `.changes` file should leave it out. In this case the original source archive on the distribution site must match exactly, byte-for-byte, the original source archive which was used to generate the `.dsc` file and diff which are being uploaded.

4.2.13. `Standards-Version`

The most recent version of the standards (the `dpkg` programmers' and policy manuals and associated texts) with which the package complies. This is updated manually when editing the source package to conform to newer standards; it can sometimes be used to tell when a package needs attention.

Its format is the same as that of a version number except that no epoch or Debian revision is allowed - see 'Version numbering', page 26.

4.2.14. `Distribution`

In a `.changes` file or parsed changelog output this contains the (space-separated) name(s) of the distribution(s) where this version of the package should be or was installed. Distribution names follow the rules for package names. (See '`Package`', page 19).

Current distribution values are `stable`, `unstable`, `contrib`, `non-free` and `experimental`.

4.2.15. `Urgency`

This is a description of how important it is to upgrade to this version from previous ones. It consists of a single keyword usually taking one of the values `LOW`, `MEDIUM` or `HIGH`) followed by an optional commentary (separated by a space) which is usually in parentheses. For example:

```
Urgency: LOW (HIGH for diversions users)
```

This field appears in the `.changes` file and in parsed changelogs; its value appears as the value of the `urgency` attribute in a `dpkg`-style changelog (see '`debian/changelog`', page 13).

Urgency keywords are not case-sensitive.

4.2.16. `Date`

In `.changes` files and parsed changelogs, this gives the date the package was built or last edited.

4.2.17. `Format`

This field occurs in `.changes` files, and specifies a format revision for the file. The format described here is version `1.5`. The syntax of the format value is the same as that of a package version number except that no epoch or Debian revision is allowed - see 'Version numbering', page 26.

4.2.18. `Changes`

In a `.changes` file or parsed changelog this field contains the human-readable changes data, describing the differences between the last version and the current one.

There should be nothing in this field before the first newline; all the subsequent lines must be indented by at least one space; blank lines must be represented by a line consiting only of a space and a full stop.

Each version's change information should be preceded by a 'title' line giving at least the version, distribution(s) and urgency, in a human-readable way.

If data from several versions is being returned the entry for the most recent version should be returned first, and entries should be separated by the representation of a blank line (the 'title' line may also be followed by the representation of blank line).

4.2.19. `Filename` and `MSDOS-Filename`

These fields in `Packages` files give the filename(s) of (the parts of) a package in the distribution directories, relative to the root of the Debian hierarchy. If the package has been split into several parts the parts are all listed in order, separated by spaces.

4.2.20. `Size` and `MD5sum`

These fields in `Packages` files give the size (in bytes, expressed in decimal) and MD5 checksum of the file(s) which make(s) up a binary package in the distribution. If the package is split into several parts the values for the parts are listed in order, separated by spaces.

4.2.21. `Status`

This field in `dpkg`'s status file records whether the user wants a package installed, removed or left alone, whether it is broken (requiring reinstallation) or not and what its current state on the system is. Each of these pieces of information is a single word.

4.2.22. `Config-Version`

If a package is not installed or not configured, this field in `dpkg`'s status file records the last version of the package which was successfully configured.

4.2.23. `Conffiles`

This field in `dpkg`'s status file contains information about the automatically-managed configuration files held by a package. This field should *not* appear anywhere in a package!

4.2.24. Obsolete fields

These are still recognised by **dpkg** but should not appear anywhere any more.

Revision
Package-Revision
Package_Revision
: The Debian revision part of the package version was at one point in a separate control file field. This field went through several names.

Recommended
: Old name for **Recommends**

Optional
: Old name for **Suggests**.

Class
: Old name for **Priority**.

5. Version numbering

Every package has a version number, in its `Version` control file field.

`dpkg` imposes an ordering on version numbers, so that it can tell whether packages are being up- or downgraded and so that `dselect` can tell whether a package it finds available is newer than the one installed on the system. The version number format has the most significant parts (as far as comparison is concerned) at the beginning.

The version number format is: [*epoch*`:`]*upstream-version*[`-`*debian-revision*].

The three components here are:

epoch
: This is a single unsigned integer, which should usually be small. It may be omitted, in which case zero is assumed. If it is omitted then the *upstream-version* may not contain any colons.

  It is provided to allow mistakes in the version numbers of older versions of a package, and also a package's previous version numbering schemes, to be left behind.

  `dpkg` will not usually display the epoch unless it is essential (non-zero, or if the *upstream-version* contains a colon); `dselect` does not display epochs at all in the main part of the package selection display.

upstream-version
: This is the main part of the version. It is usually version number of the original ('upstream') package of which the `.deb` file has been made, if this is applicable. Usually this will be in the same format as that specified by the upstream author(s); however, it may need to be reformatted to fit into `dpkg`'s format and comparison scheme.

  The comparison behaviour of `dpkg` with respect to the *upstream-version* is described below. The *upstream-version* portion of the version number is mandatory.

  The *upstream-version* may contain only alphanumerics and the characters `+ . - :` (full stop, plus, hyphen, colon) and should start with a digit. If there is no *debian-revision* then hyphens are not allowed; if there is no *epoch* then colons are not allowed.

debian-revision
: This part of the version represents the version of the modifications that were made to the package to make it a Debian binary package. It is in the same format as the *upstream-version* and `dpkg` compares it in the same way.

  It is optional; if it isn't present then the *upstream-version* may not contain a hyphen. This format represents the case where a piece of software was written specifically to be turned into a Debian binary package, and so there is only one 'debianization' of it and therefore no revision indication is required.

  It is conventional to restart the *debian-revision* at `1` each time the *upstream-version* is increased.

  `dpkg` will break the *upstream-version* and *debian-revision* apart at the last hyphen in the string. The absence of a *debian-revision* compares earlier than the presence of one (but note that the *debian-revision* is the least significant part of the version number).

> The *debian-revision* may contain only alphanumerics and the characters `+` and `.` (plus and full stop).

The *upstream-version* and *debian-revision* parts are compared by `dpkg` using the same algorithm:

The strings are compared from left to right.

First the initial part of each string consisting entirely of non-digit characters is determined. These two parts (one of which may be empty) are compared lexically. If a difference is found it is returned. The lexical comparison is a comparison of ASCII values modified so that all the letters sort earlier than all the non-letters.

Then the initial part of the remainder of each string which consists entirely of digit characters is determined. The numerical values of these two parts are compared, and any difference found is returned as the result of the comparison. For these purposes an empty string (which can only occur at the end of one or both version strings being compared) counts as zero.

These two steps are repeated (chopping initial non-digit strings and initial digit strings off from the start) until a difference is found or both strings are exhausted.

Note that the purpose of epochs is to allow us to leave behind mistakes in version numbering, and to cope with situations where the version numbering changes. It is *not* there to cope with version numbers containing strings of letters which `dpkg` cannot interpret (such as `ALPHA` or `pre-`), or with silly orderings (the author of this manual has heard of a package whose versions went `1.1`, `1.2`, `1.3`, `1`, `2.1`, `2.2`, `2` and so forth).

If an upstream package has problematic version numbers they should be converted to a sane form for use in the `Version` field.

6. Package maintainer scripts and installation procedure

6.1. Introduction to package maintainer scripts

It is possible supply scripts as part of a package which `dpkg` will run for you when your package is installed, upgraded or removed.

These scripts should be the files `preinst`, `postinst`, `prerm` and `postrm` in the control area of the package. They must be proper exectuable files; if they are scripts (which is recommended) they must start with the usual `#!` convention. They should be readable and executable to anyone, and not world-writeable.

`dpkg` looks at the exit status from these scripts. It is important that they exit with a non-zero status if there is an error, so that `dpkg` can stop its processing. For shell scripts this means that you *almost always* need to use `set -e` (this is usually true when writing shell scripts, in fact). It is also important, of course, that they don't exit with a non-zero status if everything went well.

It is necessary for the error recovery procedures that the scripts be idempotent: ie, invoking the same script several times in the same situation should do no harm. If the first call failed, or aborted half way through for some reason, the second call should merely do the things that were left undone the first time, if any, and exit with a success status.

When a package is upgraded a combination of the scripts from the old and new packages is called in amongst the other steps of the upgrade procedure. If your scripts are going to be at all complicated you need to be aware of this, and may need to check the arguments to your scripts.

Broadly speaking the `preinst` is called before (a particular version of) a package is installed, and the `postinst` afterwards; the `prerm` before (a version of) a package is removed and the `postrm` afterwards.

6.2. Summary of ways maintainer scripts are called

- *new-preinst* `install`
- *new-preinst* `install` *old-version*
- *new-preinst* `upgrade` *old-version*
- *old-preinst* `abort-upgrade` *new-version*

- *postinst* `configure` *most-recently-configured-version*
- *old-postinst* `abort-upgrade` *new version*
- *conflictor's-postinst* `abort-remove in-favour` *package new-version*
- *deconfigured's-postinst* `abort-deconfigure in-favour` *failed-install-package version* `removing` *conflicting-package version*

- *prerm* `remove`
- *old-prerm* `upgrade` *new-version*
- *new-prerm* `failed-upgrade` *old-version*
- *conflictor's-prerm* `remove in-favour` *package new-version*

- *deconfigured's-prerm* **`deconfigure in-favour`** *package-being-installed version* **`removing`** *conflicting-package version*

- *postrm* **`remove`**
- *postrm* **`purge`**
- *old-postrm* **`upgrade`** *new-version*
- *new-postrm* **`failed-upgrade`** *old-version*
- *new-postrm* **`abort-install`**
- *new-postrm* **`abort-install`** *old-version*
- *new-postrm* **`abort-upgrade`** *old-version*
- *disappearer's-postrm* **`disappear`** *overwriter new-version*

6.3. Details of unpack phase of installation or upgrade

The procedure on installation/upgrade/overwrite/disappear (ie, when running **`dpkg --unpack`**, or the unpack stage of **`dpkg --install`**) is as follows. In each case if an error occurs the actions in are general run backwards - this means that the maintainer scripts are run with different arguments in reverse order. These are the 'error unwind' calls listed below.

1. i. If a version the package is already installed, call
   old–prerm **`upgrade`** *new–version*

   ii. If this gives an error (ie, a non-zero exit status), dpkg will attempt instead:
   new–prerm **`failed-upgrade`** *old–version*
   Error unwind, for both the above cases:
   old–postinst **`abort-upgrade`** *new–version*

2. If a 'conflicting' package is being removed at the same time:

   i. If any packages depended on that conflicting package and **`--auto-deconfigure`** is specified, call, for each such package:
   deconfigured's–prerm **`deconfigure \`**
   `in-favour` *package–being–installed version* **`\`**
   `removing` *conflicting–package version*
   Error unwind:
   deconfigured's–postinst **`abort-deconfigure \`**
   `in-favour` *package–being–installed–but–failed version* **`\`**
   `removing` *conflicting–package version*
   The deconfigured packages are marked as requiring configuration, so that if **`--install`** is used they will be configured again if possible.

   ii. To prepare for removal of the conflicting package, call:
   conflictor's–prerm **`remove in-favour`** *package new–version*
   Error unwind:
   conflictor's–postinst **`abort-remove \`**
   `in-favour` *package new–version*

3. i. If the package is being upgraded, call:
   new–preinst **`upgrade`** *old–version*

ii. Otherwise, if the package had some configuration files from a previous version installed (ie, it is in the 'configuration files only' state):
    new–preinst `install` *old–version*

iii. Otherwise (ie, the package was completely purged):
    new–preinst `install`
    Error unwind versions, respectively:
    new–postrm `abort-upgrade` *old–version*
    new–postrm `abort-install` *old–version*
    new–postrm `abort-install`

4. The new package's files are unpacked, overwriting any that may be on the system already, for example any from the old version of the same package or from another package (backups of the old files are left around, and if anything goes wrong dpkg will attempt to put them back as part of the error unwind).

   It is an error for a package to contains files which are on the system in another package, unless `Replaces` is used (see '`Replaces` - overwriting files and replacing packages', page 40). Currently the `--force-overwrite` flag is enabled, downgrading it to a warning, but this may not always be the case.

   Packages which overwrite each other's files produce behaviour which though deterministic is hard for the system administrator to understand. It can easily lead to 'missing' programs if, for example, a package is installed which overwrites a file from another package, and is then removed again.[1]

5. i. If the package is being upgraded, call
    old–postrm `upgrade` *new–version*

   ii. If this fails, `dpkg` will attempt:
    new–postrm `failed-upgrade` *old–version*
    Error unwind, for both cases:
    old–preinst `abort-upgrade` *new–version*

   This is the point of no return - if `dpkg` gets this far, it won't back off past this point if an error occurs. This will leave the package in a fairly bad state, which will require a successful reinstallation to clear up, but it's when `dpkg` starts doing things that are irreversible.

6. Any files which were in the old version of the package but not in the new are removed.

7. The new file list replaces the old.

8. The new maintainer scripts replace the old.

9. Any packages all of whose files have been overwritten during the installation, and which aren't required for dependencies, are considered to have been removed. For each such package,

   i. `dpkg` calls:
    disappearer's–postrm `disappear \`
        overwriter overwriter–version

[1] Part of the problem is due to what is arguably a bug in `dpkg`.

ii. The package's maintainer scripts are removed.

iii. It is noted in the status database as being in a sane state, namely not installed (any conffiles it may have are ignored, rather than being removed by `dpkg`). Note that disappearing packages do not have their prerm called, because `dpkg` doesn't know in advance that the package is going to vanish.

10. Any files in the package we're unpacking that are also listed in the file lists of other packages are removed from those lists. (This will lobotomise the file list of the 'conflicting' package if there is one.)

11. The backup files made during installation, above, are deleted.

12. The new package's status is now sane, and recorded as 'unpacked'. Here is another point of no return - if the conflicting package's removal fails we do not unwind the rest of the installation; the conflicting package is left in a half-removed limbo.

13. If there was a conflicting package we go and do the removal actions (described below), starting with the removal of the conflicting package's files (any that are also in the package being installed have already been removed from the conflicting package's file list, and so do not get removed now).

6.4. Details of configuration

When we configure a package (this happens with `dpkg --install`, or with `--configure`), we first update the conffiles and then call:
postinst `configure` *most–recently–configured–version*

No attempt is made to unwind after errors during configuration.

If there is no most recently configured version `dpkg` will pass a null argument; older versions of dpkg may pass `<unknown>` (including the angle brackets) in this case. Even older ones do not pass a second argument at all, under any circumstances.

6.5. Details of removal and/or configuration purging

1. *prerm* `remove`

2. The package's files are removed (except conffiles).

3. *postrm* `remove`

4. All the maintainer scripts except the postrm are removed.

   If we aren't purging the package we stop here. Note that packages which have no postrm and no conffiles are automatically purged when removed, as there is no difference except for the `dpkg` status.

5. The conffiles and any backup files (-files, `#*#` files, `%`-files, `.dpkg-{old,new,tmp}`, etc.) are

removed.

6. *postrm* `purge`

7. The package's file list is removed.

No attempt is made to unwind after errors during removal.

7. Descriptions of packages - the `Description` field

The `Description` control file field is used by `dselect` when the user is selecting which packages to install and by `dpkg` when it displays information about the status of packages and so forth. It is included on the FTP site in the `Packages` files, and may also be used by the Debian WWW pages.

The description is intended to describe the program to a user who has never met it before so that they know whether they want to install it. It should also give information about the significant dependencies and conflicts between this package and others, so that the user knows why these dependencies and conflicts have been declared.

The field's format is as follows:

    Description: *single line synopsis*
     extended description over several lines

The synopsis is often printed in lists of packages and so forth, and should be as informative as possible. Every package should also have an extended description.

7.1. Types of formatting line in the extended description

- Those starting with a single space are part of a paragraph. Successive lines of this form will be word-wrapped when displayed. The leading space will usually be stripped off.
- Those starting with two or more spaces. These will be displayed verbatim. If the display cannot be panned horizontally the displaying program will linewrap them 'hard' (ie, without taking account of word breaks). If it can they will be allowed to trail off to the right. None, one or two initial spaces may be deleted, but the number of spaces deleted from each line will be the same (so that you can have indenting work correctly, for example).
- Those containing a single space followed by a single full stop character. These are rendered as blank lines. This is the *only* way to get a blank line - see below.
- Those containing a space, a full stop and some more characters. These are for future expansion. Do not use them.

7.2. Notes about writing descriptions

Always start extended description lines with at least one whitespace character. Fields in the control file and in the Packages file are separated by field names starting in the first column, just as message header fields are in RFC822. Forgetting the whitespace will cause `dpkg-deb`[1] to produce a syntax error when trying to build the package. If you force it to build anyway `dpkg` will refuse to install the resulting mess.

Do not include any completely *empty* lines. These separate different records in the Packages file

[1]Version 0.93.23 or later.

and different packages in the `debian/control` file, and are forbidden in package control files. See the previous paragraph for what happens if you get this wrong.

The single line synopsis should be kept brief - certainly under 80 characters. `dselect` displays between 25 and 49 characters without panning if you're using an 80-column terminal, depending on what display options are in effect.

Do not include the package name in the synopsis line. The display software knows how to display this already, and you do not need to state it. Remember that in many situations the user may only see the synopsis line - make it as informative as you can.

The extended description should describe what the package does and how it relates to the rest of the system (in terms of, for example, which subsystem it is which part of).

The blurb that comes with a program in its announcements and/or `README` files is rarely suitable for use in a description. It is usually aimed at people who are already in the community where the package is used. The description field needs to make sense to anyone, even people who have no idea about any of the things the package deals with.

Put important information first, both in the synopis and extended description. Sometimes only the first part of the synopsis or of the description will be displayed. You can assume that there will usually be a way to see the whole extended description.

You may include information about dependencies and so forth in the extended description, if you wish.

Do not use tab characters. Their effect is not predictable.

Do not try to linewrap the summary (the part on the same line as the field name `Description`) into the extended description. This will not work correctly when the full description is displayed, and makes no sense where only the summary is available.

7.3. Example description in control file for Smail

```
Package: smail
Version: 3.1.29.1-13
Maintainer: Ian Jackson <iwj10@cus.cam.ac.uk>
Recommends: pine | mailx | elm | emacs | mail-user-agent
Suggests: metamail
Depends: cron, libc5
Conflicts: sendmail
Provides: mail-transport-agent
Description: Electronic mail transport system.
 Smail is the recommended mail transport agent (MTA) for Debian.
 .
 An MTA is the innards of the mail system - it takes messages from
 user-friendly mailer programs and arranges for them to be delivered
 locally or passed on to other systems as required.
 .
 In order to make use of it you must have one or more user level
 mailreader programs such as elm, pine, mailx or Emacs (which has Rmail
 and VM as mailreaders) installed.  If you wish to send messages other
 than just to other users of your system you must also have appropriate
 and VM as mailreaders) installed.  If you wish to send messages other
 than just to other users of your system you must also have appropriate
 networking support, in the form of IP or UUCP.
```

8. Declaring relationships between packages

Packages can declare in their control file that they have certain relationships to other packages - for example, that they may not be installed at the same time as certain other packages, and/or that they depend on the presence of others, or that they should overwrite files in certain other packages if present.

This is done using the `Depends`, `Recommends`, `Suggests`, `Conflicts`, `Provides` and `Replaces` control file fields.

8.1. Syntax of relationship fields

These fields all have a uniform syntax. They are a list of package names separated by commas.

In `Depends`, `Recommends`, `Suggests` and `Pre-Depends` (the fields which declare dependencies of the package in which they occur on other packages) these package names may also be lists of alternative package names, separated by vertical bar symbols `|` (pipe symbols).

All the fields except `Provides` may restrict their applicability to particular versions of each named package. This is done in parentheses after each individual package name; the parentheses should contain a relation from the list below followed by a version number, in the format described in 'Version numbering', page 26.

The relations allowed are `<<`, `<=`, `=`, `>=` and `>>` for strictly earlier, earlier or equal, exactly equal, later or equal and strictly later, respectively. The forms `<` and `>` were used to mean earlier/later or equal, rather than strictly earlier/later, so they should not appear in new packages (though `dpkg` still supports them).

Whitespace may appear at any point in the version specification, and must appear where it's necessary to disambiguate; it is not otherwise significant. For consistency and in case of future changes to `dpkg` it is recommended that a single space be used after a version relationship and before a version number; it is usual also to put a single space after each comma, on either side of each vertical bar, and before each open parenthesis.

For example:

```
Package: metamail
Version: 2.7-3
Depends: libc5 (>= 5.2.18-4), mime-support, csh | tcsh
```

8.2. Dependencies - `Depends`, `Recommends`, `Suggests`, `Pre-Depends`

These four fields are used to declare a dependency by one package on another. They appear in the depending package's control file.

All but `Pre-Depends` (discussed below) take effect *only* when a package is to be configured. They do not prevent a package being on the system in an unconfigured state while its dependencies are unsatisfied, and it is possible to replace a package whose dependencies are satisfied and which is properly installed with a different version whose dependencies are not and cannot be satisfied;

when this is done the depending package will be left unconfigured (since attempts to configure it will give errors) and will not function properly.

For this reason packages in an installation run are usually all unpacked first and all configured later; this gives later versions of packages with dependencies on later versions of other packages the opportunity to have their dependencies satisfied.

Thus `Depends` allows package maintainers to impose an order in which packages should be configured.

`Depends`
: This declares an absolute dependency.

  `dpkg` will not configure packages whose dependencies aren't satisfied. If it is asked to make an installation which would cause an installed package's dependencies to become unsatisfied it will complain[1], unless `--auto-deconfigure` is specified, in which case those packages will be deconfigured before the installation proceeds.

  `dselect` makes it hard for the user to select packages for installation, removal or upgrade in a way that would mean that packages' `Depends` fields would be unsatisfied. The user can override this if they wish, for example if they know that `dselect` has an out-of-date view of the real package relationships.

  The `Depends` field should be used if the depended-on package is required for the depending package to provide a significant amount of functionality.

`Recommends`
: This declares a strong, but not absolute, dependency.

  `Recommends` is ignored by `dpkg`, so that users using the command-line (who are presumed to know what they're doing) will not be impeded.

  It is treated by `dselect` exactly as `Depends` is; this makes it hard for the user to select things so as to leave `Recommends` fields unsatisfied, but they are able to do so by being persistent.

  The `Recommends` field should list packages that would be found together with this one in all but unusual installations.

`Suggests`
: This is used to declare that one package may be more useful with one or more others. Using this field tells the packaging system and the user that the listed packages are be related to this one and can perhaps enhance its usefulness, but that installing this one without them is perfectly reasonable.

  `dselect` will offer suggsted packages to the system administrator when they select the suggesting package, but the default is not to install the suggested package.

`Pre-Depends`
: This field is like `Depends`, except that it also forces `dpkg` to complete installation of the packages named before even starting the installation of the package which declares the predependency.

  `dselect` checks for predependencies when it is doing an installation run, and will attempt to find the packages which are required to be installed first and do so in the right order.

[1]Current versions (1.2.4) of `dpkg` have a bug in this area which will cause some of these problems to be ignored.

However, this process is slow (because it requires repeated invocations of `dpkg`) and troublesome (because it requires guessing where to find the appropriate files).

For these reasons, and because this field imposes restrictions on the order in which packages may be unpacked (which can be difficult for installations from multipart media, for example), `Pre-Depends` should be used sparingly, preferably only by packages whose premature upgrade or installation would hamper the ability of the system to continue with any upgrade that might be in progress.

When the package declaring it is being configured, a `Pre-Dependency` will be considered satisfied only if the depending package has been correctly configured, just as if an ordinary `Depends` had been used.

However, when a package declaring a predependency is being unpacked the predependency can be satisfied even if the depended-on package(s) are only unpacked or half-configured, provided that they have been configured correctly at some point in the past (and not removed or partially removed since). In this case both the previously-configured and currently unpacked or half-configured versions must satisfy any version clause in the `Pre-Depends` field.

8.2.1. Dependencies on shared libraries

The dependency fields listed above are used by packages which need shared libraries to declare dependencies on the appropriate packages.

These dependencies are usually determined automatically using `dpkg-shlibdeps` and inserted in the package control file using the control file substitution variables mechanism; see ‘`debian/substvars` and variable substitutions’, page 15 and ‘Tools for processing source packages’, page 8.

8.2.2. Deconfiguration due to removal during bulk installations

If `dpkg` would like to remove a package due to a conflict, as described above, but this would violate a dependency of some other package on the system, `dpkg` will usually not remove the conflicting package and halt with an error.

However, if the `--auto-deconfigure` (`-B`) option is used `dpkg` will automatically ‘deconfigure’ the package with the problematic dependency, so that the conflicting package can be removed and the package we’re trying to install can be installed. If `dpkg` is being asked to install packages (rather than just unpacking them) it will try to reconfigure the package when it has unpacked all its arguments, in the hope that one of the other packages it is installing will satisfy the problematic dependency.

`dselect` supplies this argument to `dpkg` when it invokes it, so that bulk installations proceed smoothly.

8.3. Alternative packages - `Conflicts` and `Replaces`

When one package declares a conflict with another `dpkg` will refuse to allow them to be installed on the system at the same time.

If one package is to be installed, the other must be removed first - if the package being installed is marked as replacing ('Replaces - overwriting files and replacing packages', page 40) the one on the system, or the one on the system is marked as deselected, or both packages are marked **Essential**, then **dpkg** will automatically remove the package which is causing the conflict, otherwise it will halt the installation of the new package with an error.

dselect makes it hard to select conflicting packages, though the user can override this if they wish. If they do not override it then **dselect** will select one of the packages for removal, and the user must make sure it is the right one. In the future **dselect** will look for the presence of a **Replaces** field to help decide which package should be installed and which removed.

A package will not cause a conflict merely because its configuration files are still installed; it must be at least half-installed.

A special exception is made for packages which declare a conflict with their own package name, or with a virtual package which they provide (see below): this does not prevent their installation, and allows a package to conflict with others providing a replacement for it. You use this feature when you want the package in question to be the only package providing something.

A **Conflicts** entry should almost never have an 'earlier than' version clause. This would prevent **dpkg** from upgrading or installing the package which declared such a conflict until the upgrade or removal of the conflicted-with package had been completed. This aspect of installation ordering is not handled by **dselect**, so that the use **Conflicts** in this way is likely to cause problems for 'bulk run' upgrades and installations.

8.4. Virtual packages - Provides

As well as the names of actual ('concrete') packages, the package relationship fields **Depends**, **Recommends**, **Suggests** and **Conflicts** may mention virtual packages.

A virtual package is one which appears in the **Provides** control file field of another package. The effect is as if the package(s) which provide a particular virtual package name had been listed by name everywhere were the virtual package name appears.

If there are both a real and a virtual package of the same name then the dependency may be satisfied (or the conflict caused) by either the real package or any of the virtual packages which provide it. This is so that, for example, supposing we have

```
Package: vm
Depends: emacs
```

and someone else releases an xemacs package they can say

```
Package: xemacs
Provides: emacs
```

and all will work in the interim (until a purely virtual package name is decided on and the **emacs** and **vm** packages are changed to use it).

If a dependency or a conflict has a version number attached then only real packages will be considered to see whether the relationship is satisfied (or the prohibition violated, for a conflict) - it is assumed that a real package which provides virtual package is not of the 'right' version. So, a **Provides** field may not contain version numbers, and the version number of the concrete package which provides a particular virtual package will not be looked at when considering a dependency on or conflict with the virtual package name.

If you want to specify which of a set of real packages should be the default to satisfy a particular dependency on a virtual package, you should list the real package as alternative before the virtual.

8.5. `Replaces` - overwriting files and replacing packages

The `Replaces` control file field has two purposes, which come into play in different situations.

Virtual packages ('Virtual packages - `Provides`', page 39) are not considered when looking at a `Replaces` field - the packages declared as being replaced must be mentioned by their real names.

8.5.1. Overwriting files in other packages

Firstly, as mentioned before, it is usually an error for a package to contains files which are on the system in another package, though currently the `--force-overwrite` flag is enabled by default, downgrading the error to a warning,

If the overwriting package declares that it replaces the one containing the file being overwritten then `dpkg` will proceed, and replace the file from the old package with that from the new. The file will no longer be listed as 'owned' by the old package.

If a package is completely replaced in this way, so that `dpkg` does not know of any files it still contains, it is considered to have disappeared. It will be marked as not wanted on the system (selected for removal) and not installed. Any conffiles details noted in the package will be ignored, as they will have been taken over by the replacing package(s). The package's `postrm` script will be run to allow the package to do any final cleanup required. See 'Summary of ways maintainer scripts are called', page 28.

In the future `dpkg` will discard files which overwrite those from another package which declares that it replaces the one being installed (so that you can install an older version of a package without problems).

This usage of `Replaces` only takes effect when both packages are at least partially on the system at once, so that it can only happen if they do not conflict or if the conflict has been overridden.

8.5.2. Replacing whole packages, forcing their removal

Secondly, `Replaces` allows `dpkg` and `dselect` to resolve which package should be removed when a conflict - see 'Alternative packages - `Conflicts` and `Replaces`', page 38. This usage only takes effect when the two packages *do* conflict, so that the two effects do not interfere with each other.

8.6. Defaults for satisfying dependencies - ordering

Ordering is significant in dependency fields.

Usually dselect will suggest to the user that they select the package with the most 'fundamental' class (eg, it will prefer Base packages to Optional ones), or the one that they 'most wanted' to select in some sense.

In the absence of other information `dselect` will offer a default selection of the first named package in a list of alternatives.

However, there is no way to specify the ‘order’ of several packages which all provide the same thing, when that thing is listed as a dependency.

Therefore a dependency on a virtual package should contain a concrete package name as the first alternative, so that this is the default.

For example, consider the set of packages:

```
Package: glibcdoc
Recommends: info-browser

Package: info
Provides: info-browser

Package: emacs
Provides: info-browser
```

If `emacs` and `info` both have the same priority then `dselect`’s choice is essentially random. Better would be

```
Package: glibcdoc
Recommends: info | info-browser
```

so that `dselect` defaults to selecting the lightweight standalone info browser.

9. Configuration file handling

`dpkg` can do a certain amount of automatic handling of package configuration files.

Whether this mechanism is appropriate depends on a number of factors, but basically there are two approaches to any particular configuration file.

The easy method is to ship a best-effort configuration in the package, and use `dpkg`'s conffile mechanism to handle updates. If the user is unlikely to want to edit the file, but you need them to be able to without losing their changes, and a new package with a changed version of the file is only released infrequently, this is a good approach.

The hard method is to build the configuration file from scratch in the `postinst` script, and to take the responsibility for fixing any mistakes made in earlier versions of the package automatically. This will be appropriate if the file is likely to need to be different on each system.

9.1. Automatic handling of configuration files by `dpkg`

A package may contain a control area file called `conffiles`. This file should be a list of filenames of configuration files needing automatic handling, separated by newlines. The filenames should be absolute pathnames, and the files referred to should actually exist in the package.

When a package is upgraded `dpkg` will process the configuration files during the configuration stage, shortly before it runs the package's `postinst` script,

For each file it checks to see whether the version of the file included in the package is the same as the one that was included in the last version of the package (the one that is being upgraded from); it also compares the version currently installed on the system with the one shipped with the last version.

If neither the user nor the package maintainer has changed the file, it is left alone. If one or the other has changed their version, then the changed version is preferred - ie, if the user edits their file, but the package maintainer doesn't ship a different version, the user's changes will stay, silently, but if the maintainer ships a new version and the user hasn't edited it the new version will be installed (with an informative message). If both have changed their version the user is prompted about the problem and must resolve the differences themselves.

The comparisons are done by calculating the MD5 message digests of the files, and storing the MD5 of the file as it was included in the most recent version of the package.

When a package is installed for the first time `dpkg` will install the file that comes with it, unless that would mean overwriting a file already on the filesystem.

However, note that `dpkg` will *not* replace a conffile that was removed by the user (or by a script). This is necessary because with some programs a missing file produces an effect hard or impossible to achieve in another way, so that a missing file needs to be kept that way if the user did it.

Note that a package should *not* modify a `dpkg`-handled conffile in its maintainer scripts. Doing this will lead to `dpkg` giving the user confusing and possibly dangerous options for conffile update when the package is upgraded.

9.2. Fully-featured maintainer script configuration handling

For files which contain site-specific information such as the hostname and networking details and so forth, it is better to create the file in the package's `postinst` script.

This will typically involve examining the state of the rest of the system to determine values and other information, and may involve prompting the user for some information which can't be obtained some other way.

When using this method there are a couple of important issues which should be considered:

If you discover a bug in the program which generates the configuration file, or if the format of the file changes from one version to the next, you will have to arrange for the postinst script to do something sensible - usually this will mean editing the installed configuration file to remove the problem or change the syntax. You will have to do this very carefully, since the user may have changed the file, perhaps to fix the very problem that your script is trying to deal with - you will have to detect these situations and deal with them correctly.

If you do go down this route it's probably a good idea to make the program that generates the configuration file(s) a separate program in `/usr/sbin`, by convention called *package*`config` and then run that if appropriate from the post-installation script. The *package*`config` program should not unquestioningly overwrite an existing configuration - if its mode of operation is geared towards setting up a package for the first time (rather than any arbitrary reconfiguration later) you should have it check whether the configuration already exists, and require a `--force` flag to overwrite it.

10. Alternative versions of an interface - `update-alternatives`

When several packages all provide different versions of the same program or file it is useful to have the system select a default, but to allow the system administrator to change it and have their decisions respected.

For example, there are several versions of the `vi` editor, and there is no reason to prevent all of them from being installed at once, each under their own name (`nvi`, `vim` or whatever). Nevertheless it is desirable to have the name `vi` refer to something, at least by default.

If all the packages involved cooperate, this can be done with `update-alternatives`.

Each package provides its own version under its own name, and calls `update-alternatives` in its postinst to register its version (and again in its prerm to deregister it).

See the manpage `update-alternatives(8)` for details.

If `update-alternatives` does not seem appropriate you may wish to consider using diversions instead.

11. Diversions - overriding a package's version of a file

It is possible to have **dpkg** not overwrite a file when it reinstalls the package it belongs to, and to have it put the file from the package somewhere else instead.

This can be used locally to override a package's version of a file, or by one package to override another's version (or provide a wrapper for it).

Before deciding to use a diversion, read 'Alternative versions of an interface - **update-alternatives**', page 44 to see if you really want a diversion rather than several alternative versions of a program.

There is a diversion list, which is read by **dpkg**, and updated by a special program **dpkg-divert**. Please see **dpkg-divert(8)** for full details of its operation.

When a package wishes to divert a file from another, it should call **dpkg-divert** in its preinst to add the diversion and rename the existing file. For example, supposing that a **smailwrapper** package wishes to install a wrapper around **/usr/sbin/smail**:

```
if [ install = "$1" ]; then
    dpkg-divert --package smailwrapper --add --rename \
                --divert /usr/sbin/smail.real /usr/sbin/smail
fi
```

Testing **$1** is necessary so that the script doesn't try to add the diversion again when **smailwrapper** is upgraded. The **--package smailwrapper** ensures that **smailwrapper**'s copy of **/usr/sbin/smail** can bypass the diversion and get installed as the true version.

The postrm has to do the reverse:

```
if [ remove = "$1" ]; then
    dpkg-divert --package smailwrapper --remove --rename \
                --divert /usr/sbin/smail.real /usr/sbin/smail
fi
```

Do not attempt to divert a file which is vitally important for the system's operation - when using **dpkg-divert** there is a time, after it has been diverted but before **dpkg** has installed the new version, when the file does not exist.

12. Shared libraries

Packages containing shared libraries must be constructed with a little care to make sure that the shared library is always available. This is especially important for packages whose shared libraries are vitally important, such as the libc.

Firstly, your package should install the shared libraries under their normal names. For example, the `libgdbm1` package should install `libgdbm.so.1.7.3` as `/usr/lib/libgdbm.so.1.7.3`. The files should not be renamed or relinked by any prerm or postrm scripts; `dpkg` will take care of renaming things safely without affecting running programs, and attempts to interfere with this are likely to lead to problems.

Secondly, your package should include the symlink that `ldconfig` would create for the shared libraries. For example, the `libgdbm1` package should include a symlink from `/usr/lib/libgdbm.so.1` to `libgdbm.so.1.7.3`. This is needed so that `ld.so` can find the library in between the time `dpkg` installs it and `ldconfig` is run in the `postinst` script. Futhermore, and *this is very important*, the symlink must be placed before the library it points to in the `.deb` file. Currently the way to ensure the ordering is done properly is to create the symlink in the appropriate `debian/tmp/.../lib` directory before installing the library when you build the package.

If you do the above your package does not need to call `ldconfig` in its maintainer scripts. It is especially important not to call `ldconfig` in the postrm or preinst scripts in the case where the package is being upgraded (see the programmer's manual), as `ldconfig` will see the temporary names that `dpkg` uses for the files while it is installing them and will make the shared library links point to them, just before `dpkg` continues the installation and removes the links!

13. Configuration of `init`

13.1. Introduction to the `init.d` scheme

The `/etc/init.d` directory contains the scripts executed by `init` when init state (or 'runlevel') is changed (see `init(8)`).

These scripts are be referenced by symbolic links in the `/etc/rc`*n*`.d` directories. When changing runlevels, init looks in the directory `/etc/rc`*n*`.d` for the scripts it should execute, where *n* is the runlevel that is being changed to.

The names of the links all have the form `S`*mmscript* or `K`*mmscript* where *mm* is a two-digit number and *script* is the name of the script (this should be the same as the name of the actual script in `/etc/init.d`. When `init` changes runlevel first the targets of the links whose names starting with a `K` are executed, each with the single argument `stop`, followed by the scripts prefixed with an `S`, each with the single argument `start`. The `K` links are responsible for killing services and the `S` link for starting services upon entering the runlevel.

For example, if we are changing from runlevel 2 to runlevel 3, init will first execute all of the `K` prefixed scripts it finds in `/etc/rc3.d`, and then all of the `S` prefixed scripts. The links starting with `K` will cause the referred-to file to be executed with an argument of `stop`, and the `S` links with an argument of `start`.

The two-digit number *mm* is used to decide which order to start and stop things in - low-numbered links have their scripts run first. For example, the `K20` scripts will be executed before the `K30` scripts. This is used when a certain service must be started before another. For example, the name server `bind` might need to be started before the news server `inn` so that `inn` can set up its access lists. In this case, the script that starts `bind` should have a lower number than the script that starts `inn` so that it runs first:

```
/etc/rc2.d/S17bind
/etc/rc2.d/S70inn
```

13.2. Writing `init.d` scripts

Packages can and should place scripts in `/etc/init.d` to start or stop services at boot time or during a change of runlevel. These scripts should be named `/etc/init.d/`*package*, and they should accept one argument, saying what to do: `start`, meaning to starts the service, or `stop`, to stop the service. Optionally they can support `reload` which causes the configuration to be reloaded.

The `init.d` scripts should ensure that they will behave sensibly if invoked with `start` when the service is already running, or with `stop` when it isn't, and that they don't kill unfortunately-named user processes. The best way to achieve this is usually to use `start-stop-daemon`.

These scripts should not fail obscurely when the configuration files remain but the package has been removed, as the default in `dpkg` is to leave configuration files on the system after the package has been removed. Only when it is executed with the `--purge` option will dpkg remove configuration files. Therefore, you should include a `test` statement at the top of the script, like this:

`test -f` *program–executed–later–in–script* `|| exit 0`

13.3. Managing the rc*n*.d links - update-rc.d

A program is provided, `update-rc.d`, to make it easier for package maintainers to arrange for the proper creation and removal of `/etc/rc`*n*`.d` symbolic links from their postinst and postrm scripts.

You should use this script to make changes to `/etc/rc`*n*`.d` and *never* include any `/etc/rc`*n*`.d` symbolic links in the actual archive.

By default `update-rc.d` will start services in each of the multi-user state runlevels (2, 3, 4, and 5) and stop them in the halt runlevel (0), the single-user runlevel (1) and the reboot runlevel (6). The system administrator will have the opportunity to customize runlevels by simply adding, moving, or removing the symbolic links in `/etc/rc`*n*`.d`.

To get the default behaviour for your package, put in your postinst script

```
update-rc.d package default >/dev/null
```

and in your postrm

```
if [ purge = "$1" ]; then
    update-rc.d package remove >/dev/null
fi
```

This will use a default sequence number of 20. If it does not matter when or in which order the script is run, use this default. If it does, then you should talk to the maintainer of the `sysvinit` package or post to `debian-devel`, and they will help you choose a number.

For more information about using `update-rc.d`, please consult its manpage `update-rc.d(8)`.

13.4. Boot-time initialisation - rc.boot

There is another directory, `/etc/rc.boot`, which contains scripts which are run once per machine boot. This facility is provided for initialisation of hardware devices, cleaning up of leftover files, and so forth.

For example, the `kbd` package provides a script here for initialising the keyboard layout and console font and mode.

The files in `/etc/rc.boot` should *not* be links into `/etc/init.d` - they should be the scripts themselves.

`rc.boot` should *not* be used for starting general-purpose daemons and similar activities. This should be done using the `rc`*n*`.d` scheme, above, so that the services can be started and stopped cleanly when the runlevel changes or the machine is to be shut down or rebooted.

13.5. Notes

Do not include the `/etc/rc`*n*`.d/*` symbolic links in the `.deb` filesystem archive! *This will cause* *problems!* You should create them with `update-rc.d`, as above.

Do not include the `/etc/rc`*n*`.d/*` symbolic links in `dpkg`'s conffiles list! *This will cause problems! Do*, however, include the `/etc/init.d` scripts in conffiles.

13.6. Example

The `bind` DNS (nameserver) package wants to make sure that the nameserver is running in multiuser runlevels, and is properly shut down with the system. It puts a script in `/etc/init.d`, naming the script appropriately `bind`. As you can see, the script interprets the argument `reload` to send the nameserver a `HUP` signal (causing it to reload its configuration); this way the user can say `/etc/init.d/bind reload` to reload the nameserver.

```
#!/bin/sh
# Original version by Robert Leslie <rob@mars.org>, edited by iwj
test -x /usr/sbin/named || exit 0
case "$1" in
  start)
    test -f /etc/named.boot -a -f /var/named/boot.options || exit 0
    start-stop-daemon --start --verbose --exec /usr/sbin/named
    ;;
  stop)
    start-stop-daemon --stop --verbose  \
        --pidfile /var/run/named.pid --exec /usr/sbin/named
    ;;
  reload)
    start-stop-daemon --stop --signal 1 --verbose  \
        --pidfile /var/run/named.pid --exec /usr/sbin/named
    ;;
  *)
    echo "Usage: /etc/init.d/bind {start|stop|reload}" >&2
    exit 1
    ;;
esac
exit 0
```

Another example on which to base your `/etc/init.d` scripts is in `/etc/init.d/skeleton`.

If this package is happy with the default setup from `update-rc.d`, namely an ordering number of 20 and having named running in all runlevels, it can say in its postinst:

```
update-rc.d bind default >/dev/null
```

And in its postrm, to remove the links when the package is purged:

```
if [ purge = "$1" ]; then
     update-rc.d acct remove >/dev/null
fi
```

14. `dselect`'s interface to its installation methods

`dselect` calls scripts from its installation methods when it needs to actually access data from the distribution. The core program `dselect` itself just calls these scripts and provides the package and access method selection interfaces. The installation methods are responsible for invoking `dpkg` as appropriate.

Each installation method has three scripts:
- Setup installation parameters.
- Update list of available packages.
- Install.

`dselect` searches for methods in `/usr/lib/dpkg/methods` and `/usr/local/lib/dpkg/methods`.

14.1. Functions of the method scripts

The setup script is run just after the user has chosen an installation method. It should prompt the user for parameters like the site to NFS-mount or FTP from, the directory to use, or the directory or filesystem where the `.deb` files can be found, or the tape or floppy device to install from. It should store the responses under `/var/lib/dpkg/methods` - see below. If no available packages list is available it should perhaps offer to scan the available packages.

The update script should obtain a list of available packages if possible, and run `dpkg --update-avail`, `dpkg --merge-avail` and/or `dpkg --forget-old-unavail` to load it into `dpkg` and `dselect`'s database of available packages. If no packages list was available and the user was offered and accepted the option of scanning the actual files available this scan should be done here, using `dpkg --record-avail`.

The install script should feed all the available `.deb` files to `dpkg --iGOEB` (this is equivalent to `dpkg --install --refuse-downgrade --selected-only --skip-same-version --auto-deconfigure`). The `-R` (`--recursive`) option for traversing subdirectories may also be useful here).

If any of these scripts needs to display a message for the user, it should wait for the user to hit 'return' before exiting so that dselect doesn't immediately rewrite the screen.

If a method script succeeds (returns a zero exit status) `dselect` will return immediately to the main menu, with the 'next' option highlighted ready for the user to select it. If it fails `dselect` will display a message and wait for the user to hit return.

14.2. Location and arguments of the method scripts

A set of scripts (henceforth known as a group) may provide several methods on the 'main menu' with different behaviour. For example, there might be a generic get-packages-by-FTP group which might provide methods in the main menu for installation directly from one of the Debian mirror sites as well as for installation from a user-specified site.

Each group of methods implemented by the same set of scripts should have a subdirectory `/usr/lib/dpkg/methods/`*group* or `/usr/local/lib/dpkg/methods/`*group*, containing:

`names`

a list of user-visible methods provided by these scripts.

`setup`
`update`
`install`
executable programs, the scripts themselves.

`desc.`*option*
description file.

`names` will be formatted as a list of lines, each containing:
sequence method summary

sequence is a two-digit number that will be used much like **`rc.d`** prefixes to control the order in the main menu. If in doubt use 50.

method is a name which is displayed by **`dselect`** as the name of the method, and which will be passed to **`setup`**, **`update`** and **`unpack`** as their first argument.

summary is the brief description string for **`dselect`**'s menu.

Each of the three scripts gets the same three arguments: *vardir*, *group* and *method*. *vardir* is the base directory for storing **`dpkg`** and **`dselect`**'s state, usually **`/var/lib/dpkg`**; this is passed in so that the **`--admindir`** option to **`dselect`** is honoured).

Each option may have an extended description in **`desc.`***option*. This should be formatted like the extended description part of a **`Description`** field entry *shifted one character to the left*.

*vardir***`/methods`** will exist, and a method group may use a *vardir***`/methods/`***group* directory to store its state.

The group name and method name must follow the rules for C identifiers.