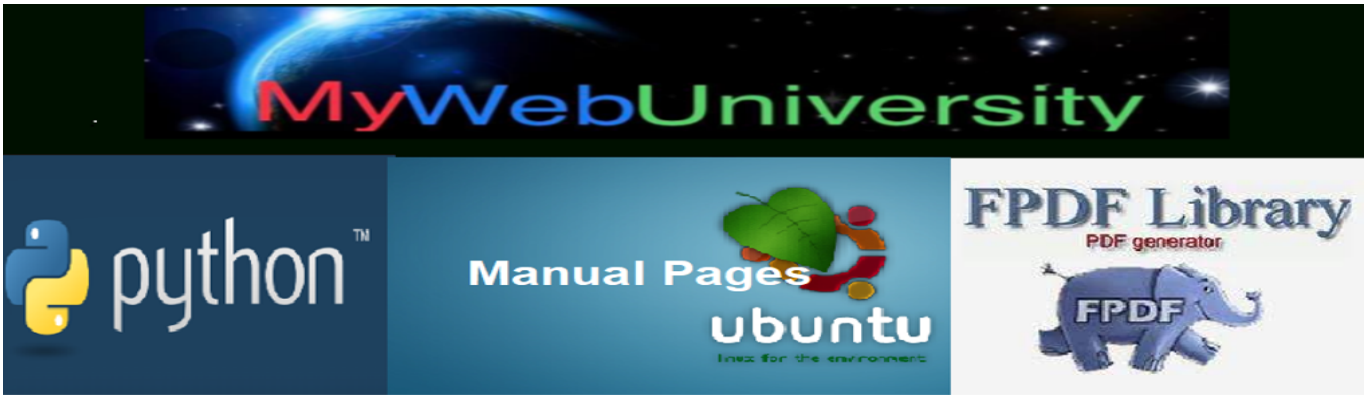

Rocky Enterprise Linux 9.2 Manual Pages on command 'dpkg-gensymbols.1'

$ man dpkg-gensymbols.1

dpkg-gensymbols(1) dpkg suite dpkg-gensymbols(1)

NAME

dpkg-gensymbols - generate symbols files (shared library dependency information)

SYNOPSIS

dpkg-gensymbols [option...]

DESCRIPTION

dpkg-gensymbols scans a temporary build tree (debian/tmp by default) looking for libraries and generates a symbols file describing them. This file, if non-empty, is then installed in the DEBIAN subdirectory of the build tree so that it ends up included in the control information of the package.

When generating those files, it uses as input some symbols files provided by the maintainer. It looks for the following files (and uses the first that is found):

? debian/package.symbols.arch

? debian/symbols.arch

? debian/package.symbols

? debian/symbols

The main interest of those files is to provide the minimal version associated to each symbol provided by the libraries. Usually it corresponds to the first version of that package that provided the symbol, but it can be manually incremented by the maintainer if the ABI of the symbol is extended without breaking backwards compatibility. It's the responsibility of the maintainer to keep those files up-to-date and accurate, but dpkg-gensymbols helps with that.

When the generated symbols files differ from the maintainer supplied one, dpkg-gensymbols

will print a diff between the two versions. Furthermore if the difference is too significant, it will even fail (you can customize how much difference you can tolerate, see the -c option).

MAINTAINING SYMBOLS FILES

The base interchange format of the symbols file is described in deb-symbols(5), which is used by the symbols files included in binary packages. These are generated from template symbols files with a format based on the former, described in deb-src-symbols(5) and included in source packages.

The symbols files are really useful only if they reflect the evolution of the package through several releases. Thus the maintainer has to update them every time that a new symbol is added so that its associated minimal version matches reality.

The diffs contained in the build logs can be used as a starting point, but the maintainer, additionally, has to make sure that the behaviour of those symbols has not changed in a way that would make anything using those symbols and linking against the new version, stop working with the old version.

In most cases, the diff applies directly to the debian/package.symbols file. That said, further tweaks are usually needed: it's recommended for example to drop the Debian revision from the minimal version so that backports with a lower version number but the same upstream version still satisfy the generated dependencies. If the Debian revision can't be dropped because the symbol really got added by the Debian specific change, then one should suffix the version with ?~?.

Before applying any patch to the symbols file, the maintainer should double-check that it's sane. Public symbols are not supposed to disappear, so the patch should ideally only add new lines.

Note that you can put comments in symbols files.

Do not forget to check if old symbol versions need to be increased. There is no way dpkg-gensymbols can warn about this. Blindly applying the diff or assuming there is nothing to change if there is no diff, without checking for such changes, can lead to packages with loose dependencies that claim they can work with older packages they cannot work with. This will introduce hard to find bugs with (partial) upgrades.

Good library management

A well-maintained library has the following features:

? its API is stable (public symbols are never dropped, only new public symbols are

added) and changes in incompatible ways only when the SONAME changes;

? ideally, it uses symbol versioning to achieve ABI stability despite internal changes and API extension;

? it doesn't export private symbols (such symbols can be tagged optional as workaround).

While maintaining the symbols file, it's easy to notice appearance and disappearance of symbols. But it's more difficult to catch incompatible API and ABI change. Thus the maintainer should read thoroughly the upstream changelog looking for cases where the rules of good library management have been broken. If potential problems are discovered, the upstream author should be notified as an upstream fix is always better than a Debian specific work-around.

OPTIONS

-Ppackage-build-dir

Scan package-build-dir instead of debian/tmp.

-ppackage

Define the package name. Required if more than one binary package is listed in debian/control (or if there's no debian/control file).

-vversion

Define the package version. Defaults to the version extracted from debian/changelog. Required if called outside of a source package tree.

-elibrary-file

Only analyze libraries explicitly listed instead of finding all public libraries. You can use shell patterns used for pathname expansions (see the File::Glob(3perl) manual page for details) in library-file to match multiple libraries with a single argument (otherwise you need multiple -e).

-ldirectory

Prepend directory to the list of directories to search for private shared libraries (since dpkg 1.19.1). This option can be used multiple times.

Note: Use this option instead of setting LD_LIBRARY_PATH, as that environment variable is used to control the run-time linker and abusing it to set the shared library paths at build-time can be problematic when cross-compiling for example.

-Ifilename

Use filename as reference file to generate the symbols file that is integrated in the package itself.

-O[filename]

Print the generated symbols file to standard output or to filename if specified, rather than to debian/tmp/DEBIAN/symbols (or package-build-dir/DEBIAN/symbols if -P was used). If filename is pre-existing, its contents are used as basis for the generated symbols file. You can use this feature to update a symbols file so that it matches a newer upstream version of your library.

-t Write the symbol file in template mode rather than the format compatible with deb-symbols(5). The main difference is that in the template mode symbol names and tags are written in their original form contrary to the post-processed symbol names with tags stripped in the compatibility mode. Moreover, some symbols might be omitted when writing a standard deb-symbols(5) file (according to the tag processing rules) while all symbols are always written to the symbol file template.

-c[0-4]

Define the checks to do when comparing the generated symbols file with the template file used as starting point. By default the level is 1. Increasing levels do more checks and include all checks of lower levels.

Level 0

Never fails.

Level 1

Fails if some symbols have disappeared.

Level 2

Fails if some new symbols have been introduced.

Level 3

Fails if some libraries have disappeared.

Level 4

Fails if some libraries have been introduced.

This value can be overridden by the environment variable DPKG_GENSYMBOLS_CHECK_LEVEL.

-q Keep quiet and never generate a diff between generated symbols file and the template file used as starting point or show any warnings about new/lost libraries or new/lost symbols. This option only disables informational output but not the checks themselves (see -c option).

-aarch

Assume arch as host architecture when processing symbol files. Use this option to

generate a symbol file or diff for any architecture provided its binaries are already available.

-d Enable debug mode. Numerous messages are displayed to explain what dpkg-gensymbols does.

-V Enable verbose mode. The generated symbols file contains deprecated symbols as comments. Furthermore in template mode, pattern symbols are followed by comments listing real symbols that have matched the pattern.

-?, --help

Show the usage message and exit.

--version

Show the version and exit.

ENVIRONMENT

DPKG_GENSYMBOLS_CHECK_LEVEL

Overrides the command check level, even if the -c command-line argument was given (note that this goes against the common convention of command-line arguments having precedence over environment variables).

DPKG_COLORS

Sets the color mode (since dpkg 1.18.5). The currently accepted values are: auto (default), always and never.

DPKG_NLS

If set, it will be used to decide whether to activate Native Language Support, also known as internationalization (or i18n) support (since dpkg 1.19.0). The accepted values are: 0 and 1 (default).

SEE ALSO

<https://people.redhat.com/drepper/symbol-versioning>, <https://people.redhat.com/drepper/goodpractice.pdf>, <https://people.redhat.com/drepper/dsohowto.pdf>, deb-src-symbol(5), deb-symbols(5), dpkg-shlibdeps(1).