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 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: any line with '#' as the first character is a comment except if it starts with '#include' (see section **Using includes**). Lines starting with '#MISSING:' are special comments documenting symbols that have disappeared.

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.

Using #PACKAGE# substitution

In some rare cases, the name of the library varies between architectures. To avoid hardcoding the name of the package in the symbols file, you can use the marker *#PACKAGE#*. It will be replaced by the real package name during installation of the symbols files. Contrary to the

#MINVER# marker, *#PACKAGE#* will never appear in a symbols file inside a binary package.

Using symbol tags

Symbol tagging is useful for marking symbols that are special in some way. Any symbol can have an arbitrary number of tags associated with it. While all tags are parsed and stored, only some of them are understood by **dpkg-gensymbols** and trigger special handling of the symbols. See subsection **Standard symbol tags**for reference of these tags.

Tag specification comes right before the symbol name (no whitespace is allowed in between). It always starts with an opening bracket **(**, ends with a closing bracket **)** and must contain at least one tag. Multiple tags are separated by the **|** character. Each tag can optionally have a value which is separated form the tag name by the **=** character. Tag names and values can be arbitrary strings except they cannot contain any of the special **) | =** characters. Symbol names following a tag specification can optionally be quoted with either **'** or characters to allow whitespaces in them. However, if there are no tags specified for the symbol, quotes are treated as part of the symbol name which continues up until the first space.

(tag1=i am marked|tag name with space)tagged quoted symbol@Base 1.0 (optional)tagged_unquoted_symbol@Base 1.0 1 untagged_symbol@Base 1.0

The first symbol in the example is named *tagged quoted symbol* and has two tags: *tag1* with value *i am marked* and *tag name with space* that has no value. The second symbol named *tagged_unquoted_symbol* is only tagged with the tag named *optional*. The last symbol is an example of the normal untagged symbol.

Since symbol tags are an extension of the **deb-symbols(5)** format, they can only be part of the symbols files used in source packages (those files should then be seen as templates used to build the symbols files that are embedded in binary packages). When **dpkg-gensymbols** is called without the **-t** option, it will output symbols files compatible to the **deb-symbols(5)** format: it fully processes symbols according to the requirements of their standard tags and strips all tags from the output. On the contrary, in template mode (**-t**) all symbols and their tags (both standard and unknown ones) are kept in the output and are written in their original form as they were loaded.

Standard symbol tags

optional

A symbol marked as optional can disappear from the library at any time and that will never cause **dpkg-gensymbols** to fail. However, disappeared optional symbols will continuously appear as MISSING in the diff in each new package revision. This behaviour serves as a reminder for the maintainer that such a symbol needs to be removed from the symbol file or readded to the library. When the optional symbol, which was previously declared as MISSING, suddenly reappears in the next revision, it will be upgraded back to the existing status with its minimum version unchanged.

This tag is useful for symbols which are private where their disappearance do not cause ABI breakage. For example, most of C++ template instantiations fall into this category. Like any other tag, this one may also have an arbitrary value: it could be used to indicate why the symbol is considered optional.

arch=*architecture list*

This tag allows one to restrict the set of architectures where the symbol is supposed to exist. When the symbols list is updated with the symbols discovered in the library, all arch-specific symbols which do not concern the current host architecture are treated as if they did not exist. If an arch-specific symbol matching the current host architecture does not exist in the library, normal procedures for missing symbols apply and it may cause **dpkg-gensymbols** to fail. On the other hand, if the arch-specific symbol is found when it was not supposed to exist (because the current host architecture is not listed in the tag), it is made arch neutral (i.e. the arch tag is dropped and the symbol will appear in the diff due to this change), but it is not considered as new.

When operating in the default non-template mode, among arch-specific symbols only

those that match the current host architecture are written to the symbols file. On the contrary, all arch-specific symbols (including those from foreign arches) are always written to the symbol file when operating in template mode.

The format of *architecture list*is the same as the one used in the **Build-Depends** field of *debian/control* (except the enclosing square brackets []). For example, the first symbol from the list below will be considered only on alpha, any-amd64 and ia64 architectures, the second only on linux architectures, while the third one anywhere except on armel.

(arch=alpha any-amd64 ia64)a_64bit_specific_symbol@Base 1.0 (arch=linux-any)linux_specific_symbol@Base 1.0 (arch=!armel)symbol_armel_does_not_have@Base 1.0

ignore-blacklist
dpkg-gensymbols has an internal blacklist of symbols that should not appear in symbols files as they are usually only side-effects of implementation details of the toolchain. If for some reason, you really want one of those symbols to be included in the symbols file, you should tag the symbol with **ignore-blacklist**. It can be necessary for some low level toolchain libraries like libgcc.

c++ Denotes *c++* symbol pattern. See **Using symbol patterns**subsection belo w.

symver
Denotes *symver* (symbol version) symbol pattern. See **Using symbol patterns**subsection below.

regex Denotes *regex* symbol pattern. See **Using symbol patterns**subsection belo w.

Using symbol patterns

Unlike a standard symbol specification, a pattern may cover multiple real symbols from the library. **dpkg-gensymbols** will attempt to match each pattern against each real symbol that does *not* have a specific symbol counterpart defined in the symbol file. Whenever the first matching pattern is found, all its tags and properties will be used as a basis specification of the symbol. If none of the patterns matches, the symbol will be considered as new.

A pattern is considered lost if it does not match any symbol in the library. By default this will trigger a **dpkg-gensymbols** failure under **-c1** or higher level. However, if the failure is undesired, the pattern may be marked with the *optional* tag. Then if the pattern does not match anything, it will only appear in the diff as MISSING. Moreover, like any symbol, the pattern may be limited to the specific architectures with the *arch* tag. Please refer to **Standard symbol tags**subsection above for more information.

Patterns are an extension of the **deb-symbols(5)** format hence they are only valid in symbol file templates. Pattern specification syntax is not any different from the one of a specific symbol. However, symbol name part of the specification serves as an expression to be matched against *name@version* of the real symbol. In order to distinguish among different pattern types, a pattern will typically be tagged with a special tag.

At the moment, **dpkg-gensymbols** supports three basic pattern types:

c++
This pattern is denoted by the *c++* tag. It matches only C++ symbols by their demangled symbol name (as emitted by **c++filt**(1) utility). This pattern is very handy for matching symbols which mangled names might vary across different architectures while their demangled names remain the same. One group of such symbols is *non-virtual thunks* which have architecture specific offsets embedded in their mangled names. A common instance of this case is a virtual destructor which under diamond inheritance needs a non-virtual thunk symbol. For example, even if _ZThn8_N3NSB6ClassDD1Ev@Base on 32bit architectures will probably be _ZThn16_N3NSB6ClassDD1Ev@Base on 64bit ones, it can be matched with a single *c++* pattern:

libdummy.so.1 libdummy1 #MINVER# [...] (c++)non-virtual thunk to

NSB::ClassD::~ClassD()@Base 1.0 [...]

The demangled name above can be obtained by executing the following command:

$ echo '_ZThn8_N3NSB6ClassDD1Ev@Base' | c++filt

Please note that while mangled name is unique in the library by definition, this is not necessarily true for demangled names. A couple of distinct real symbols may have the same demangled name. For example, that's the case with non-virtual thunk symbols in complex inheritance configurations or with most constructors and destructors (since g++ typically generates two real symbols for them). However, as these collisions happen on the ABI level, they should not degrade quality of the symbol file.

symver

This pattern is denoted by the *symver* tag. Well maintained libraries have versioned symbols where each version corresponds to the upstream version where the symbol got added. If that's the case, you can use a *symver* pattern to match any symbol associated to the specific version. For example:

libc.so.6 libc6 #MINVER# (symver)GLIBC_2.0 2.0 [...] (symver)GLIBC_2.7 2.7 access@GLIBC_2.0 2.2

All symbols associated with versions GLIBC_2.0 and GLIBC_2.7 will lead to minimal version of 2.0 and 2.7 respectively with the exception of the symbol access@GLIBC_2.0. The latter will lead to a minimal dependency on libc6 version 2.2 despite being in the scope of the (symver)GLIBC_2.0 pattern because specific symbols take precedence over patterns.

Please note that while old style wildcard patterns (denoted by *@version in the symbol name field) are still supported, they have been deprecated by new style syntax (symver|optional)version. For example, *@GLIBC_2.0 2.0 should be written as (symver|optional)GLIBC_2.0 2.0 if the same behaviour is needed.

regex

Regular expression patterns are denoted by the *regex* tag. They match by the perl regular expression specified in the symbol name field. A regular expression is matched as it is, therefore do not forget to start it with the ^ character or it may match any part of the real symbol *name@version* string. For example:

libdummy.so.1 libdummy1 #MINVER# (regex)^mystack_.*@Base$ 1.0 (regex|optional)private 1.0

Symbols like mystack_new@Base, mystack_push@Base, mystack_pop@Base etc. will be matched by the first pattern while e.g. ng_mystack_new@Base won't. The second pattern will match all symbols having the string private in their names and matches will inherit *optional* tag from the pattern.

Basic patterns listed above can be combined where it makes sense. In that case, they are processed in the order in which the tags are specified. For example, both

(c++|regex)^NSA::ClassA::Private::privmethodd(int)@Base 1.0 (regex|c++)N3NSA6ClassA7Private11privmethoddEi@Base 1.0

will match symbols _ZN3NSA6ClassA7Private11privmethod1Ei@Base and _ZN3NSA6ClassA7Private11privmethod2Ei@Base. When matching the first pattern, the raw symbol is first demangled as C++ symbol, then the demangled name is matched against the regular expression. On the other hand, when matching the second pattern, regular expression is matched against the raw symbol name, then the symbol is tested if it is C++ one by attempting to demangle it. A failure of any basic pattern will result in the failure of the whole pattern. Therefore, for example, __N3NSA6ClassA7Private11privmethoddEi@Base will not match either of the patterns because it is not a valid C++ symbol.

In general, all patterns are divided into two groups: aliases (basic *c++* and *symver*) and generic patterns (*regex*, all combinations of multiple basic patterns). Matching of basic alias-based

patterns is fast (O(1)) while generic patterns are O(N) (N - generic pattern count) for each symbol. Therefore, it is recommended not to overuse generic patterns.

When multiple patterns match the same real symbol, aliases (first *c++*, then *symver*) are preferred over generic patterns. Generic patterns are matched in the order they are found in the symbol file template until the first success. Please note, however, that manual reordering of template file entries is not recommended because **dpkg-gensymbols** generates diffs based on the alphanumerical order of their names.

Using includes

When the set of exported symbols differ between architectures, it may become inefficient to use a single symbol file. In those cases, an include directive may prove to be useful in a couple of ways:

- You can factorize the common part in some external file and include that file in your *package*.symbols.*arch* file by using an include directive like this:

  #include *packages*.symbols.common

- The include directive may also be tagged like any symbol:

  (tag|...|tagN)#include file-to-include

  As a result, all symbols included from *file-to-include* will be considered to be tagged with *tag ... tagN* by default. You can use this feature to create a common *package*.symbols file which includes architecture specific symbol files:

  common_symbol1@Base 1.0 (arch=amd64 ia64 alpha)#include package.symbols.64bit (arch=!amd64 !ia64 !alpha)#include package.symbols.32bit common_symbol2@Base 1.0

The symbols files are read line by line, and include directives are processed as soon as they are encountered. This means that the content of the included file can override any content that appeared before the include directive and that any content after the directive can override anything contained in the included file. Any symbol (or even another #include directive) in the included file can specify additional tags or override values of the inherited tags in its tag specification. However, there is no way for the symbol to remove any of the inherited tags.

An included file can repeat the header line containing the SONAME of the library. In that case, it overrides any header line previously read. However, in general it's best to avoid duplicating header lines. One way to do it is the following:

#include libsomething1.symbols.common arch_specific_symbol@Base 1.0

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

-P*package-build-dir*

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

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

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

-e*library-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**).

-I*filename*
: 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).

-a*arch*
: 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.

SEE ALSO

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